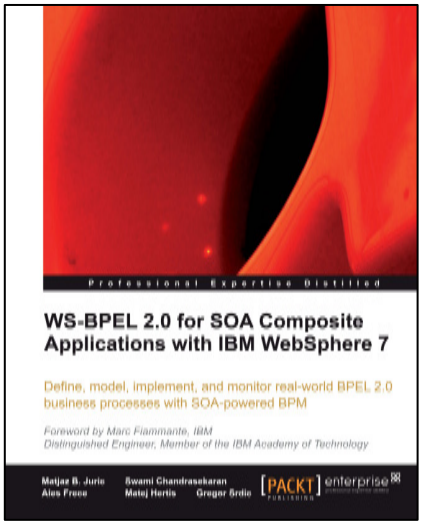
[PACKT] enterprise
professional expertise distilled
PUBLISHING

# WS-BPEL 2.0 for SOA Composite Applications with IBM WebSphere 7

**Matjaz B. Juric**
**Swami Chandrasekaran**
**Ales Frece**
**Matej Hertis**
**Gregor Srdic**




## Chapter No.6
## " Securing BPEL Processes"

## In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.6 "  Securing BPEL Processes"

A synopsis of the book's content

Information on where to buy this book

# About the Authors

**Matjaz B. Juric** holds a Ph.D. in Computer and Information Science. He is a full-time Professor at the university and head of the Cloud Computing and SOA Competence Centre. He is a Java Champion and Oracle ACE Director. He has more than 15 years of work experience. He has authored/coauthored Business Process Driven SOA using BPMN and BPEL (Business Process Execution Language) for Web Services (English and French editions), BPEL Cookbook: Best Practices for SOA-based integration and composite applications development (award for the best SOA book in 2007 by SOA World Journal), SOA Approach to Integration, Professional J2EE EAI, Professional EJB, J2EE Design Patterns Applied, and .NET Serialization Handbook. He has published chapters in More Java Gems (Cambridge University Press) and in Technology Supporting Business Solutions (Nova Science Publishers). He has also published journals and magazines, such as SOA World Journal, Web Services Journal, Java Developer's Journal, Java Report, Java World, EAI Journal, TheServerSide.com, OTN, ACM journals, and presented at conferences such as OOPSLA, Java Development, XML Europe, OOW, SCI, and others. He is a reviewer, program committee member, and conference organizer. He has been involved in several large-scale projects. In cooperation with IBM Java Technology Centre, he worked on performance analysis and optimization of RMI-IIOP, an integral part of the Java platform. He is also a member of the BPEL Advisory Board.

My efforts in this book are dedicated to my family. Thanks to my friends at SOA and Cloud Computing Centre, and Packt Publishing.

**Swami Chandrasekaran** works for IBM as an Industry Executive Architect for its Software Group—Industry Solutions. He provides architectural leadership for IBM tooling/product suite and works with its global customers in delivery of best-in-class solutions. His expertise includes next-generation networks, OSS/ BSS, SOA, Dynamic BPM, and modern web-based architectures, and TM Forum Frameworx (NGOSS). He has travelled to almost 24 countries and is constantly sought after within the company for his technical leadership and client skills. His credits include technical and strategic interface with various senior executives and institutions, including Fortune 100/500 companies and international clients. He is the SME and co-lead Architect for the WebSphere Telecom Content Pack.

He has presented at several conferences, authored articles within IBM, articles featured in BearingPoint Institute for Thought Leadership and also holds several patent disclosures. He previously worked for BearingPoint and also for Ericsson Wireless Research. He lives with his wife Ramya and daughter Harshitha in Dallas, Texas. He is an avid video gamer and during his free time he likes to write at http://www. nirvacana.com . He holds a Master's in Electrical Engineering from the University of Texas, Arlington.

There are many people whom I would like to thank. I thank my IBM management team for encouraging me and allowing me the time needed to write this book. I'd like to thank all my mentors and my family members including my in-laws who have helped and guided me over the last several years. Finally, and most importantly, I thank my wife Ramya and daughter Harshitha, for encouraging me to take this immensely challenging journey and for all the weekends and time they have sacrificed so that this book could become a reality.

I dedicate this book to my parents, my gurus, and Ramya.

**Ales Frece** (ales.frece@soa.si, ales.frece@cloud.si) is a researcher at a university, where he is preparing his doctoral dissertation. He has participated in several SOA projects as a consultant and solution designer at the SOA Competence Centre and Cloud Computing Centre. He is involved in several research and applicative projects. He has published articles and attended conferences where he has presented his extensive knowledge on

BPM, SOA, and IBM WebSphere platform. He holds several IBM SOA Technology certificates and has cooperated in launching the first cloud in Slovenia.

I would like to thank Matjaz for giving me the opportunity to contribute to this book. I would also like to thank my dear wife Darja for all her love and support.

**Matej Hertis** ( matej.hertis@soa.si, matej.hertis@cloud.si) is a researcher at a university. He graduated in computer and information sciences and is now working on his doctoral thesis. His main research areas are SOA, BPM, and Cloud Computing. He has published several articles and presented in conferences. He has been involved in several IT projects as a consultant and is a vice-head project manager at SOA Competence Centre and Cloud Computing Centre. His main expertise includes BPM, SOA, and Cloud Computing, especially on IBM WebSphere platform. He holds IBM SOA Technology certificates.

I would like to thank Matjaz B. Juric for the opportunity to be a part of this book and his guidance throughout the writing process.

**Gregor Srdic** (`gregor.srdic@cloud.si`) is a researcher at a university. He has been involved in several research and applicative projects as consultant and solution designer. He is also participating at the SOA Competency Centre and Cloud Computing Centre in the fields of SOA, BPM, and Cloud Computing. His main expertise includes business process design and business monitoring with IBM WebSphere platform and cloud management with IBM WebSphere CloudBurst Appliance.

I'd like to thank Matjaz B. Juric for his mentorship and for the opportunity to be a part of this book.

# WS-BPEL 2.0 for SOA Composite Applications with IBM WebSphere 7

Business Process Execution Language (BPEL, aka WS-BPEL) has become the de facto standard for orchestrating services in SOA composite applications. BPEL reduces the gap between business requirements and applications and allows better alignment between business processes and underlying IT architecture. BPEL is for SOA what SQL is for databases. Therefore, learning BPEL is essential for successful adoption of SOA or development of composite applications. Although BPEL looks easy at first sight, it hides large potential and has many interesting advanced features that you should get familiar with in order to maximize the value of SOA.

This book provides a comprehensive and detailed coverage of BPEL, one of the center pieces of SOA. It covers basic and advanced features of BPEL 2.0 and provides several real-world examples. In addition to the BPEL specification, the book provides comprehensive coverage of BPEL support in the IBM WebSphere SOA platform including security, transactions, human workflow, process monitoring, automatic generation of BPEL from process models, dynamic processes, and many more.

## What This Book Covers

Chapter 1, Introduction to BPEL and SOA, introduces BPEL, defining its role with regard to SOA (Service-Oriented Architecture), and explaining the process-oriented approach to SOA and the role of BPEL. It also provides short descriptions of the most important BPEL servers and compares BPEL to other business process languages.

Chapter 2, Service Composition with BPEL, describes how to define a BPEL process and makes you familiar with the basic concepts of service composition with BPEL. It also defines two example BPEL processes for business travels and shows how to develop a synchronous and then an asynchronous process.

Chapter 3, Advanced BPEL, makes you familiar with the advanced concepts of BPEL, such as loops, process termination, delays, and deadline and duration expressions and also addresses fault handling, which is a very important aspect of each business process.

Chapter 4, BPEL Processes with IBM WebSphere, illustrates how to develop BPEL processes in IBM WebSphere Integration Developer and deploy and run the BPEL processes on the IBM WebSphere Process Server.

Chapter 5, Human Interactions in BPEL, describes the different approaches to human workflow support in BPEL and analyzes their relevance in practical scenarios, and discusses real-world scenarios in which BPEL and human workflow services are used. It also describes two specifications, BPEL4People and WS-HumanTask.

Chapter 6, Securing BPEL Processes, describes how to get familiar with basic security concepts on WebSphere Application Server regarding protection of BPEL processes and looks at how to secure BPEL processes, so that they can be accessed only by authenticated users.

Chapter 7, Iterative Process Development from BPMN to BPEL, describes how to achieve synchronization between a business process model and process implementation. It models the Travel Approval process and uses this process for different synchronization scenarios such as initial, technical, business, and round-trip synchronization.

Chapter 8, Monitoring Business Processes, covers the basic concepts of Business Monitoring also known as Business Activity Monitoring (BAM). It also explains how to use IBM WebSphere to monitor BPEL processes and shows how to develop a monitor model and a dashboard in WebSphere.

Chapter 9, IBM BPM Enabled by SOA: Overview, discusses the core capabilities needed for a process integration approach, IBM's SOA reference architecture, and IBM's Business Process Management platform including WebSphere Process Server and WebSphere Enterprise Service Bus. It then looks at the fundamental SOA programming model concepts and explains how these concepts apply in the context of WID/WPS/WESB.

Chapter 10, IBM BPM Enabled By SOA—BPM in the Cloud, Dynamic Processes, and Advanced Topics, discusses several topics ranging from strategy maps creation to deployment and management of solutions on top of WebSphere BPM using the various tools provided.

# 6
# Securing BPEL Processes

In this chapter, we will get familiar with basic security concepts of WebSphere Application Server regarding protection of BPEL processes. We will look at how to secure BPEL processes, so that they can be accessed only by authenticated users. In a nutshell, we will:

- Create and protect a web service export of a BPEL process by user authentication, which requires providing a username and password inside the UsernameToken of the WS-Security specification
- Configure the web service export of a BPEL process to propagate user identity to the process, so that a process instance ownership can be claimed in that user's name
- Protect a BPEL process at SCA level as a component to implement access to the process for authorized users only

## Core concepts

BPEL as a specification does not provide any security concepts that we could leverage. All security aspects are left to the BPEL engine or, in other words, to the BPEL engine wrapper.

In WebSphere, BPEL processes are implemented as SCA components. So for BPEL processes, we can leverage all security constructs that SCA architecture offers. A BPEL process can be secured on an SCA component level so that only authorized users can access it through usage of a security permission qualifier. This qualifier defines that role-specific users must be assigned for accessing a specific SCA component, in our case a BPEL process. Before such a BPEL process is deployed on the server, it is possible to map specific users to the role qualifiers. At runtime, it is possible to dynamically add or remove users to and from this role.

A very common standardized way to expose BPEL process to the outside world is through the use of web services. In WebSphere, every SCA component can be exposed as a web service through a web service export. There are four web service export types supported, depending on the communication protocol (HTTP(S) or JMS), message exchange format (SOAP v1.1 or v1.2), and Java implementation framework (JAX-WS or JAX-RPC). We will discuss these options later in this chapter.

Web services use SOAP as a message exchange format. SOAP relies on XML. The root element of a SOAP message is a `<soap:Envelope>` that contains `<soap:Header>` and `<soap:Body>`. In the body, there are usually input data for the service operation (payload). Other data (such as credentials, correlation, and others) are contained in the header.

Web services in Java can be implemented with JAX-WS (Java API for XML Web Services) or JAX-RPC (a predecessor of JAX-WS called Java API for XML-based Remote Procedure Call) API. JAX-WS uses annotations introduced in Java SE 5 to make development and deployment of web services and their clients an easier task. JAX-WS is the preferred approach to service development. In WebSphere, JAX-WS supports WS-Policy sets to configure web service behavior in a declarative way.

WS-Policy is a framework for expressing characteristics (like capabilities or requirements) of web services. It uses flexible and extensible constructs. Each policy is a collection of many policy alternatives, each containing policy assertions. Policy assertions are used to express web service characteristics. In WebSphere, specific WS-Policy policies are grouped together in policy sets, each policy set containing one or more WS-Policy policies. The main purpose of using WS-Policy in WebSphere is to specify different web service behaviors, for example, to specify different security aspects. One of the most important security specifications for web services supported in WebSphere is WS-Security.

**WS-Security** (**WSS**) is a specification for delivering end-to-end security for web services. It provides extension to SOAP (to 1.1 and 1.2 versions) for assuring message content integrity and confidentiality. WS-Security supports a variety of security models such as PKI, Kerberos, and SSL. Moreover, WS-Security supports multiple security token formats (username token, binary security token, XML token, and EncryptedData token), trusted domains, signature formats and algorithms (Exclusive XML Canonicalization, SOAP Message Normalization), and encryption technologies (symmetric and asymmetric).

A WSS username token contains a username and is extensible to include possible authentication data, such as a password and additional data to increase security like a nonce (unique identifier to prevent replay attacks) or timestamp (to prevent replay attacks by leveraging message expiration methods). A WSS binary security token provides only one XML element which contains binary data (for example, X.509 certificate or Kerberos ticket). An XML token is an abstract token that is concretized into WSS subsequent specifications. One of the tokens is an **SAML** token (**Security Assertion Markup Language**), which is used for exchanging authentication and authorization data between different security domains (identity providers and service providers). An EncryptedData token is an encrypted version of any token contained in a WSS header to provide token confidentiality.

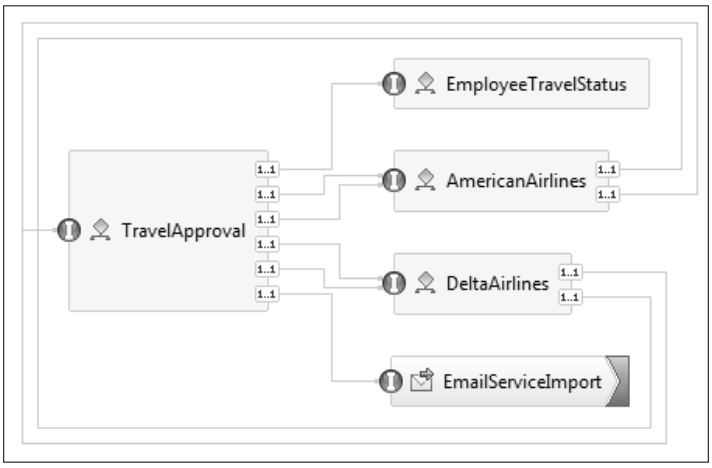
# Securing a BPEL process

We will explain how to secure a BPEL process with an example. We will expose the BPEL process as a web service. This way, a client will be able to call it in a standardized way. Next, we will create a new WS-Policy set that will require the client to provide a WS-Security header with UsernameToken containing the username and password for user authentication. We will attach this WS-Policy set to our BPEL process's web service export to protect the process. Only authenticated users will have access to the BPEL process. We will then test the example, with and without providing credentials, to see if security is working. Next, we will see that authentication information (user's identity) is not automatically propagated to the BPEL process. So, the process does not know which user called it. We will extract a user's identity from UsernameToken and propagate this identity to the BPEL process. Through another round of testing, we will see that the user who called the process will become the process instance owner. The final step in this example will be to configure the BPEL process in a way that only authenticated users, who are authorized, will be able to call it. To achieve this, we will have to define the security permission qualifier on our BPEL process, define a role that will have access to our BPEL process, and map users to this role to actually allow specific users to access our BPEL process. Later on, we will be able to add or remove users from the role to enable runtime access permission update for specific users.

# Exposing a BPEL process as web service

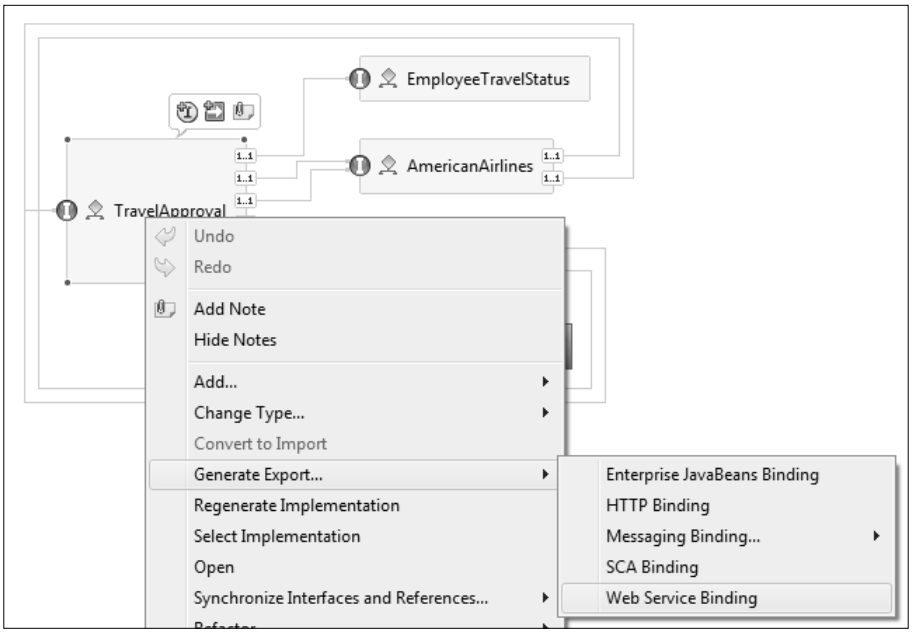
We will expose a BPEL process as a web service with the help of the following steps:

1. Let us take the Travel Approval BPEL process and open it in the WebSphere Integration Developer. The Travel Approval process is an SCA component with an interface. The Travel Approval process and surrounding components are shown in the assembly diagram represented in the following screenshot:

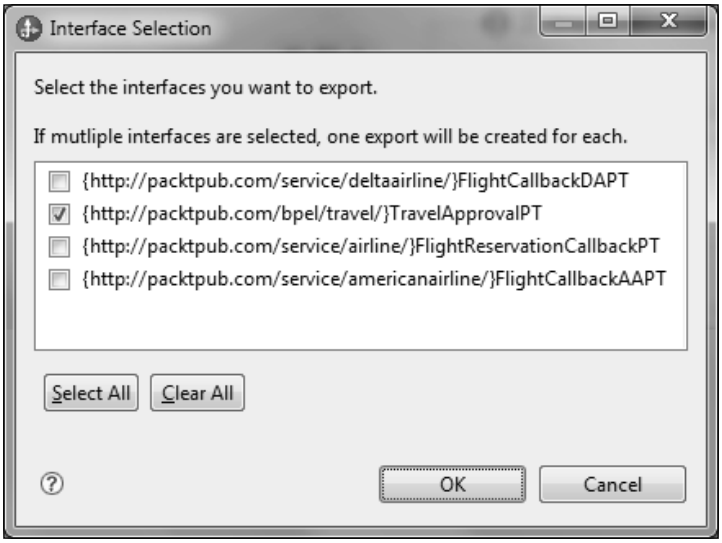



The Travel Approval process that we will use in this chapter is very similar to the process that we have developed in the previous chapter. The only difference is that this process also sends an e-mail to confirm the reservation. You can download the sample from `http://www.packtpub.com`. The sample can be deployed to the IBM WebSphere Process Server using the IBM WebSphere Integration Developer.

2. We can expose this process as a web service by generating a web service export. We can generate the web service export by right-clicking on the **TravelApproval** process and selecting **Generate Export ...** and then **Web Service Binding** from the context menu, as shown in the following screenshot:

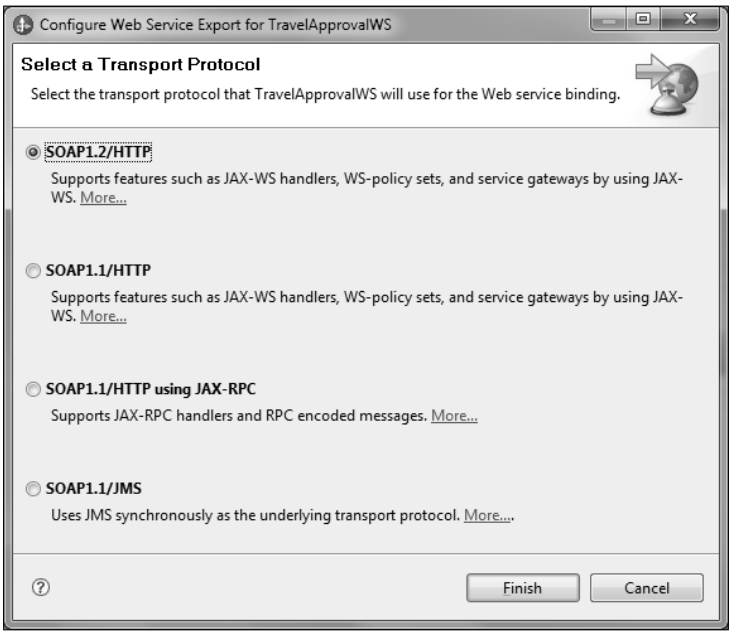



3. The Travel Approval process implements more than one interface, so a dialog asks us which interface to expose. We should select the interface that is used to run the process, that is, the **TravelApprovalPT** interface, as shown in the following screenshot:

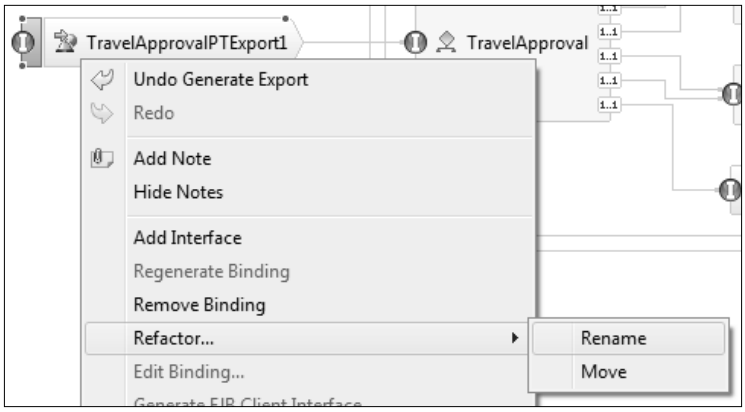


4. Next, we have to select the transport protocol. There are four options as we can see in the following image. The first two are using the HTTP protocol with JAX-WS as the implementation framework and the SOAP protocol of versions 1.1 and 1.2 respectively (the name **Simple Object Access Protocol** behind the **SOAP** abbreviation was dropped in version 1.2). SOAP 1.2 brings several advancements over SOAP 1.1. For example, it assures better interoperability, better support standards like XML Information Set (a set of definitions for use in other specifications), is truly protocol independent, and has better and more formalized extensibility. The third option uses JAX-RPC as the implementation framework for SOAP 1.1. The last option doesn't use HTTP as the transport protocol but instead uses **JMS** (**Java Message Service**).

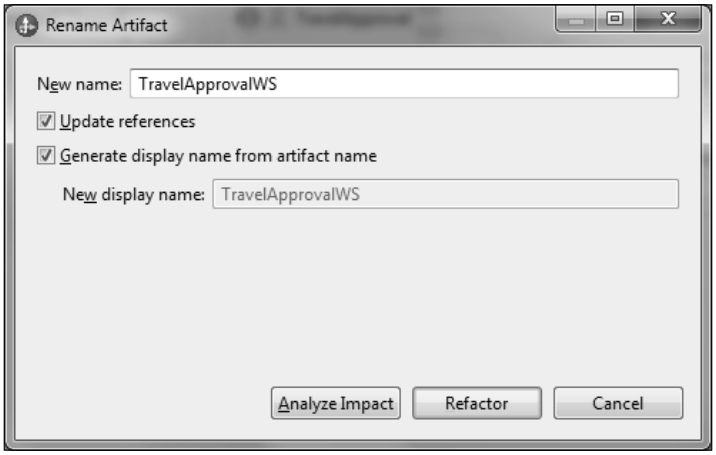
   We will use WS-Policy to define authentication rules, so we need the WS-Policy support that is offered only with JAX-WS binding. We will choose usage of SOAP messages of version 1.2, that is, **SOAP1.2/HTTP**, as the latest standard, as shown in the following screenshot. Alternatively, we could also use SOAP 1.1 with JAX-WS.




5. At this point, we need to save our assembly diagram, because we will rename the newly generated export to have a more descriptive name. We will use the refactoring option for renaming SCA components, imports, and exports. Refactoring is enabled by right-clicking on our newly created web service export, **TravelApprovalPTExport1** | **Refactor ...** | **Rename**, as shown in the following screenshot:

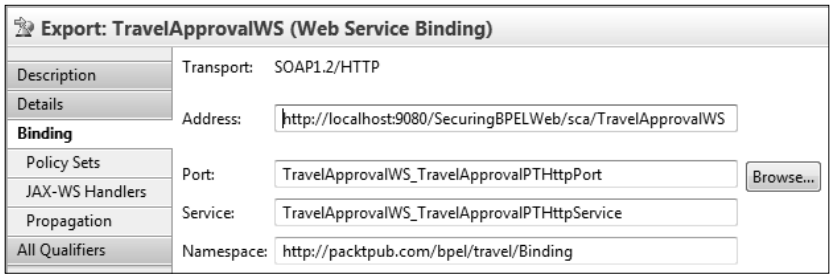



6. We choose `TravelApprovalWS` as the new name for the export and click on **Refactor**:

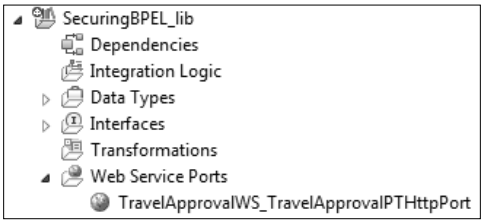



7. When we created the web service export, WebSphere Integration Developer generated the following artifacts:
   - Web service binding **TravelApprovalWS_ TravelApprovalPTHttpBinding** in namespace `http://packtpub.com/bpel/travel/Binding`
   - A service called **TravelApprovalWS_ TravelApprovalPTHttpService** with port **TravelApprovalWS_TravelApprovalPTHttpPort** and endpoint address `http://localhost:9080/SecuringBPELWeb/sca/TravelApprovalWS`

Note that the endpoint address can change when the web service is deployed on another server. We can see all these details if we select our web service export in the assembly diagram and bring up its properties by clicking on **Properties** | **Binding**:




8. All these details are actually defined in the WSDL file that was generated in our **SecuringBPEL_lib** library, which contains data types and interfaces. We can find the **TravelApprovalWS_TravelApprovalPTHttpPort** WSDL file under **Web Service Ports** in **SecuringBPEL_lib,** as shown in the following screenshot:




9. When we open this WSDL file, we can see all the content we described earlier (binding, service, port, address). Notice that the `<wsdl:import>` element imports the actual (abstract) interface from `TravelApprovalPT.wsdl`:

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions name="TravelApprovalWS_TravelApprovalPTHttp_Service" targetNamespace=
  <wsdl:import location="TravelApprovalPT.wsdl" namespace="http://packtpub.com/bpel/travel/"/>
  <wsdl:binding name="TravelApprovalWS_TravelApprovalPTHttpBinding" type="Port_0:TravelApprovalPT">
    <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="TravelApproval">
      <soap12:operation soapAction="http://packtpub.com/bpel/travel/TravelApprovalPT/TravelApproval"/>
      <wsdl:input name="TravelApprovalRequest">
        <soap12:body use="literal"/>
      </wsdl:input>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="TravelApprovalWS_TravelApprovalPTHttpService">
    <wsdl:port binding="this:TravelApprovalWS_TravelApprovalPTHttpBinding" name="TravelApprovalWS_TravelApprovalPTHttpPort">
      <soap12:address location="http://localhost:9080/Example3Web/sca/TravelApprovalWS"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

In this section, we have created a web service export for our BPEL process and examined the details that occurred behind the scenes.

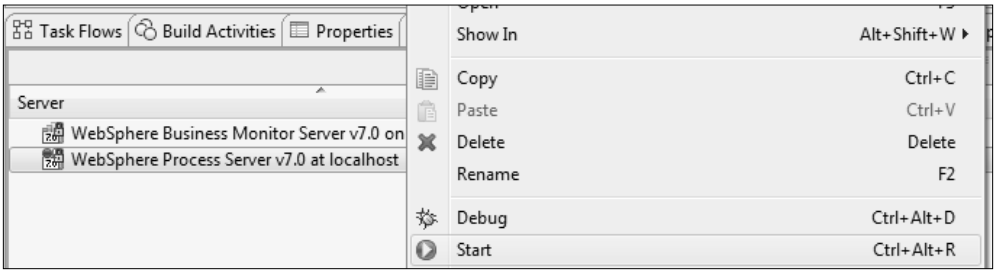
# Creating a WS-Policy set for WS-Security authentication

In this section, we will create a WS-Policy set on WebSphere Application Server through its console. The WS-Policy set will demand WS-Security **UsernameToken** to be present in the incoming SOAP Header. We will configure **UsernameToken** to be consumed, and credentials from it authenticated by the WAS users repository to determine if the caller is authentic and can call our **TravelApproval** BPEL business process.

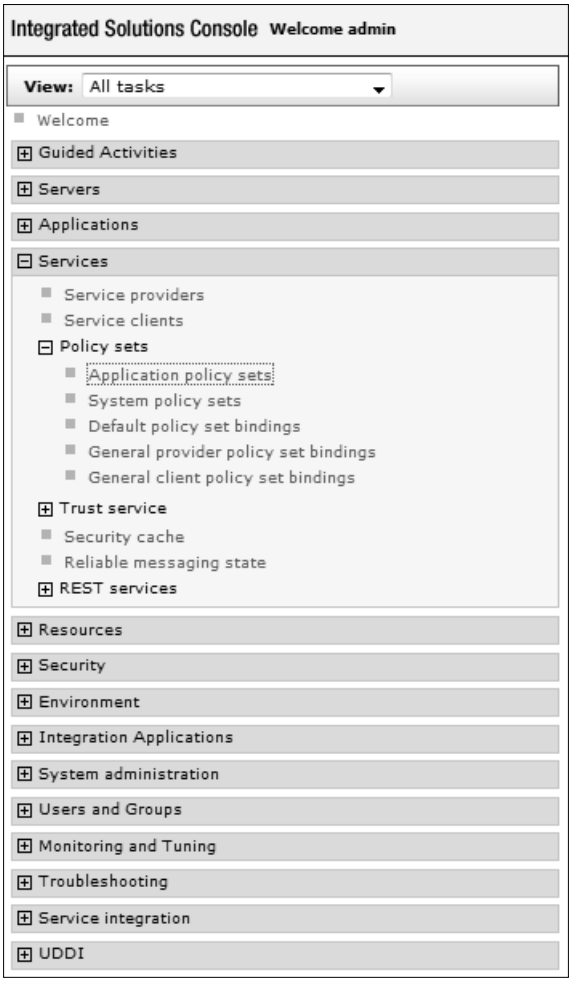
We will achieve the described level of security by completing the following steps:

1. First, we will define a security token that will be consumed at the service. There are **UsernameToken**, **X.509** certificates, **LTPA** tokens, and **Custom** tokens available, as shown in the following screenshot. **X.509** certificates require that the client send a certificate when invoking the process. An **LTPA** (**Lightweight Third-Party Authentication**) token is an IBM-specific token. It allows users to reuse their login credentials across physical servers. A **Custom** token can be defined to cover some specific requirements. We will use the **UsernameToken** in our example, because it is the most straightforward to use. Using other tokens is very similar to using **UsernameToken**, but it requires some additional work on the client side, where the client has to provide additional information (**X.509** certificate, for example)

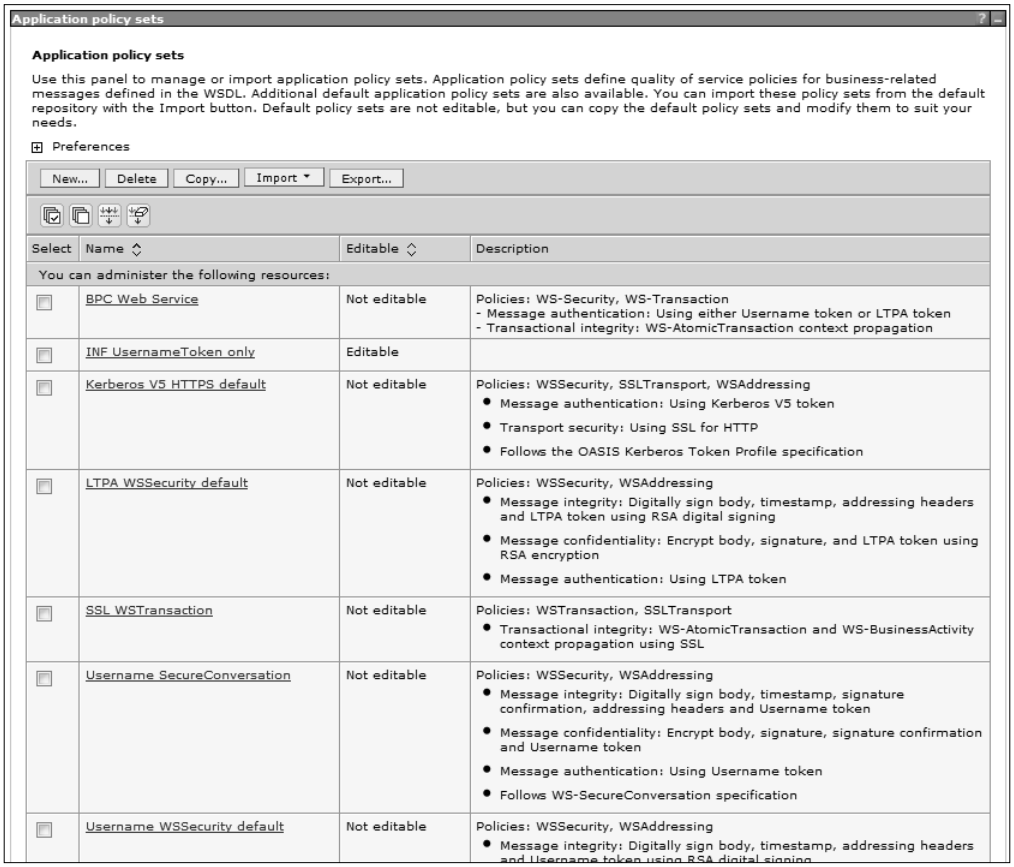

2. Next, we need to make sure that the Universal Test Environment (the WebSphere Application Server instance optimized for the development) is running. Check the **Servers** view in the WebSphere Integration Developer and if the server is in the state **Stopped**, run it by right-clicking on it and selecting **Start**:



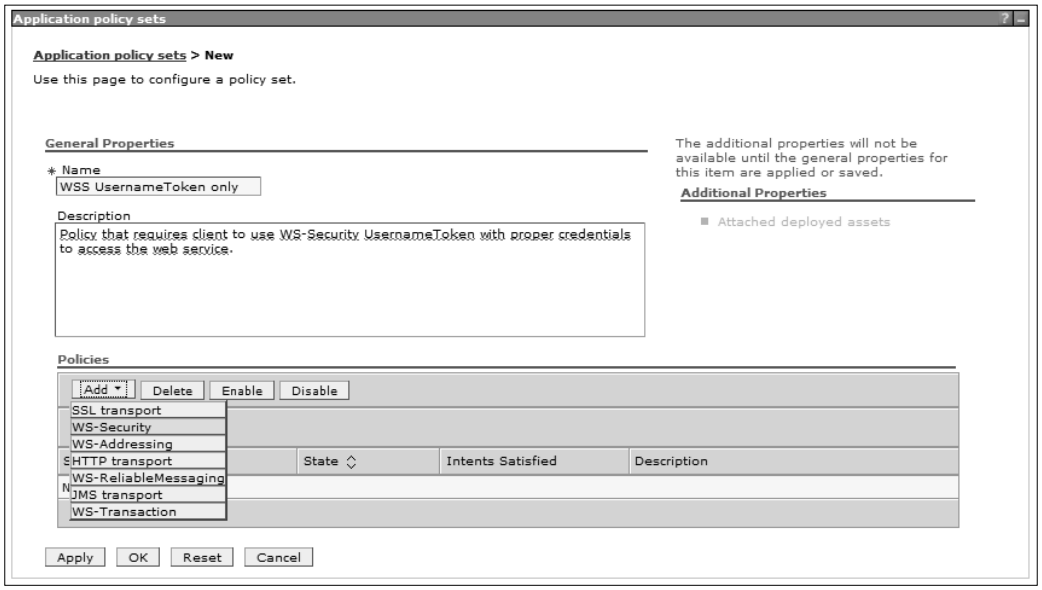
3. After the server is started, open **Integrated Solutions Console** (enter `https://HOSTNAME(localhost):PORT(9043)/ibm/console` in the browser or right-click on **started server** and from the menu select **Administration | Run administrative console**). Log in as administrator (the default administrator has username `admin` and password `admin`). In the console expand **Services | Policy sets | Application policy sets**:



4. A list of predefined policy sets appears. We will not use a predefined policy set. Rather we will create a new policy set. Click on **New...** to create a new policy set, as shown in the following screenshot:



5. To complete the creation of a new policy set, we have to specify its name and optional description. Then we have to add some policies. In our case, we will add only one policy (WS-Security) with one token definition (WS-Security **UsernameToken**). We will enter `WSS UsernameToken only` for the **Name** and `Policy that requires client to use WS-Security UsernameToken with proper credentials to access the web service` for the **Description**. Click on **Add** | **WS-Security**:

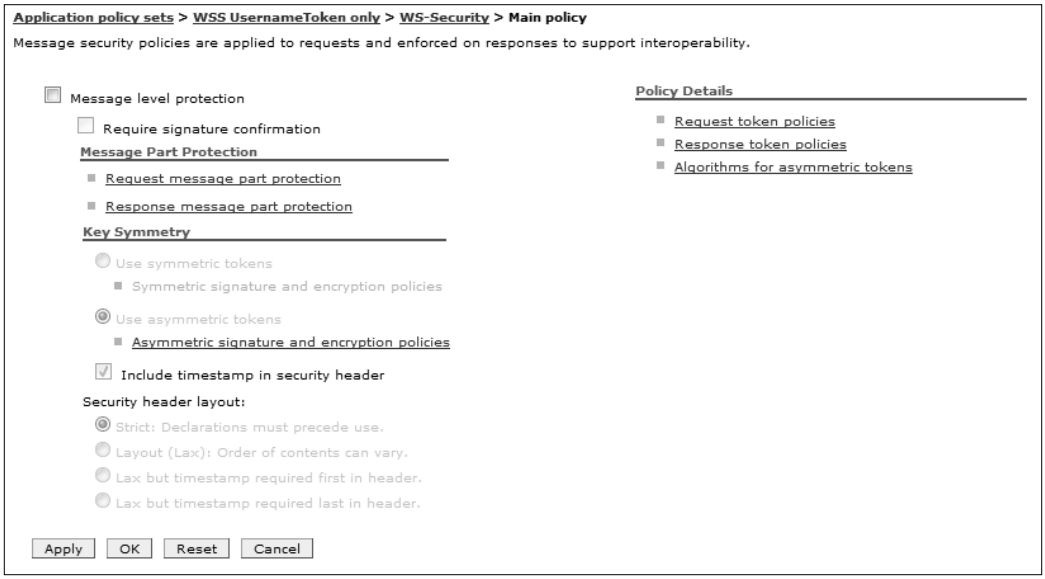



6. Optionally, we can define intents for each policy that are meant for descriptive (natural language) demands that a policy defines. Enter them in the **Intents Satisfied** field as shown in the following screenshot (enter `Require WS-Security UsernameToken` into the text box) and click on **Apply**.

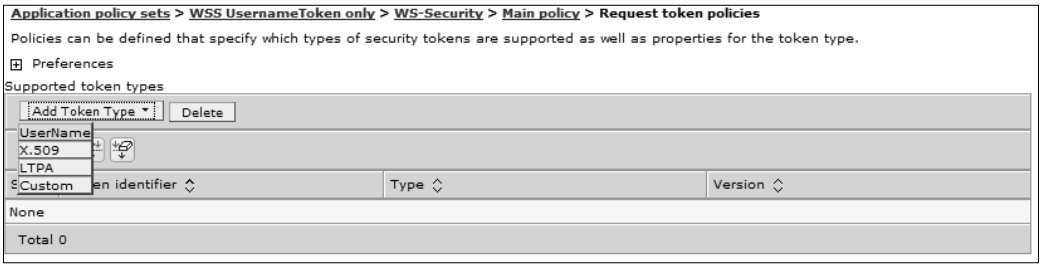
7. WS-Security is one of the policies in the policy set configuration. Each policy contains a security policy for communication with the service and the bootstrap policy for communication with the trusted service. Secure conversation bootstrap policy is used for the service consumer to acquire a security token for secure conversation from the trust service. This is achieved using a token issuing a WS-SecureConversation or WS-Trust protocol message. We can find both types of policies if we select **WS-Security** in the **Policies** list. The communication policy is called **Main policy** and the bootstrap policy is called **Secure conversation bootstrap policy**. In our case, the bootstrap policy is disabled and will remain so, because we have not defined any secure/trust requirements in the **Main policy**. We will not define them, because this is out of the scope of this example.


8. Now, we will set all the details needed for the WS-Security policy to require that the client provides the **UsernameToken**. If we select **Main policy**, we can see all available settings that concern WS-Security. By default, **Message level protection** with asymmetric tokens is enabled. This option is for using digital signatures and encryption. Details are defined in **Policy Details** section, but more on this a little later.
9. The **Require signature confirmation** option specifies whether in the response message there should be a signature confirmation token. The WS-Security v1.1 specification defines the XML element `<wsse11:SignatureConfirmation wsu:Id="…" Value="…" />`, which tells the client what happened when the service checked the request message signature. The **Value** attribute is important, because it contains the `<ds:SignatureValue>` value from the request message. If so, the request was processed as expected. On the other hand, if this attribute is not present, this tells the client that the received response is based on a request that was not signed. If the element is present but empty, this means that something went wrong and the client should proceed accordingly.
10. Under **Request message part protection** and **Response message part protection**, we can define parts of request and response messages that should be encrypted and signed. We define this with XPath expressions.

11. Under **Key Symmetry**, we can choose between symmetric and asymmetric tokens for encryption and signing. After we have chosen the appropriate tokens, we can define all the details for the selected tokens to function properly. Under **Policy Details | Algorithms for asymmetric tokens**, additional properties for asymmetric encryption are offered. Further discussion on this topic is out of the scope of this example.
12. **Security header layout** offers four options for elements ordering in the WS-Security header. A **Strict** layout means that any usage of other header elements (element referencing) must be preceded with a referenced element declaration. **Layout (Lax)** allows the order of contents in the header to vary as long as the header elements are defined within the restrictions specified in the WS-Security specification. **Lax but timestamp required first in header** and **Lax but timestamp required last in header** also pose no additional restrictions except on the position of the `<wsu:Created>` element that contains a timestamp when the message was created (first or last element in the header).
13. For our example, we will not use message-level protection, so we will uncheck the checkbox next to **Message level protection** and click on **Apply**. The result should be as shown in the following screenshot:

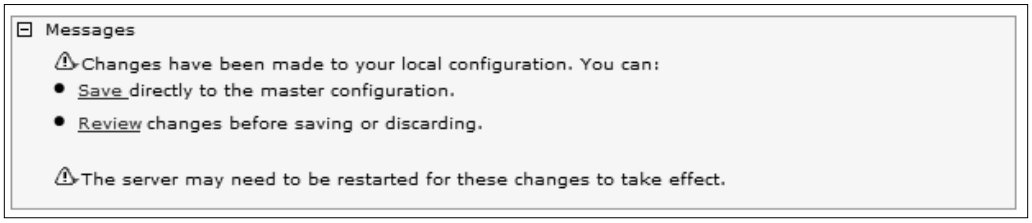



14. However, we will use **UsernameToken**. So, we have to define its usage in the request message. We will select **Request token policies** under **Policy Details** (we would choose **Response token policies** if **UsernameToken** would be required in the response message). We open the **Add Token Type** drop-down menu and select **UserName**:




15. To define the **UsernameToken** usage, we have to specify its name and select its version. For the name, we will enter `authentication_token` and select **WS-Security 1.0**. We will use the **WS-Security 1.0** specification in this example, as it defines all the necessary elements and is supported in more web service frameworks than the newer v1.1. We could have chosen **WS-Security 1.1** without any major differences for the rest of the example


16. We should now save changes to the master configuration by clicking on **Save**:

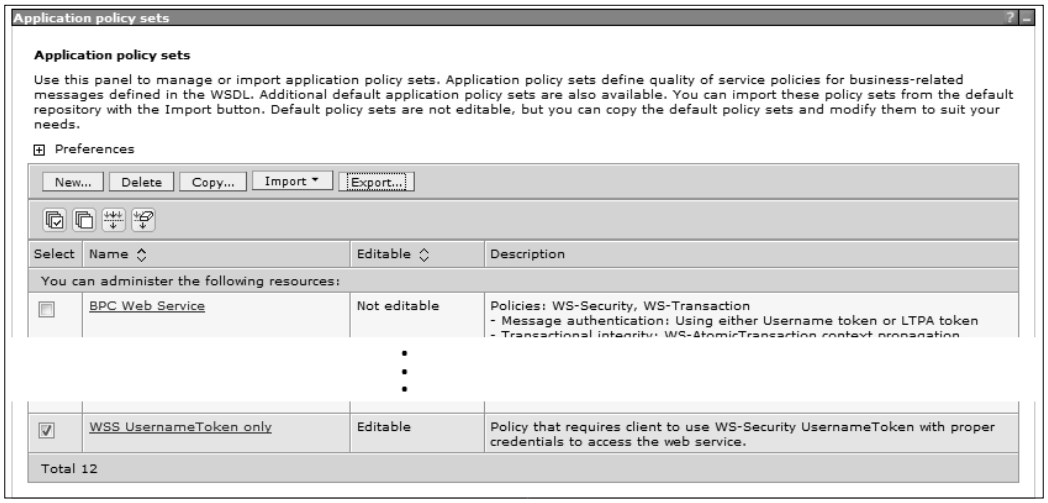



We have created a WS-Policy set that demands WS-Security **UsernameToken** to be present in the incoming SOAP Header. **UsernameToken** holds user credentials through which users are authenticated on the server. All authenticated users are then allowed to access the TravelApproval BPEL business process. All unauthenticated users cannot access the process any more.

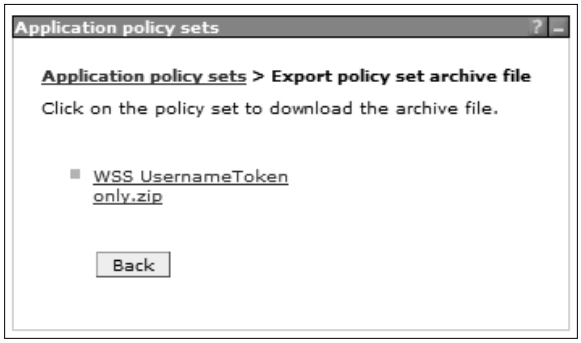
# Securing a BPEL process web service export with a WS-policy set

We have finished configuring the WS-Security policy set. Now, we will export it and import it into WebSphere Integration Developer to include it into the assembly diagram. In this way, we will be able to tell the server to apply the selected policy set on our BPEL process web service export. To use the configured WS-Security policy set on the process web service export, we have to perform the following steps:

1. First, let us export the policy set. On **Application policy sets,** click the checkbox next to **WSS UsernameToken only** policy set and click on **Export…**:

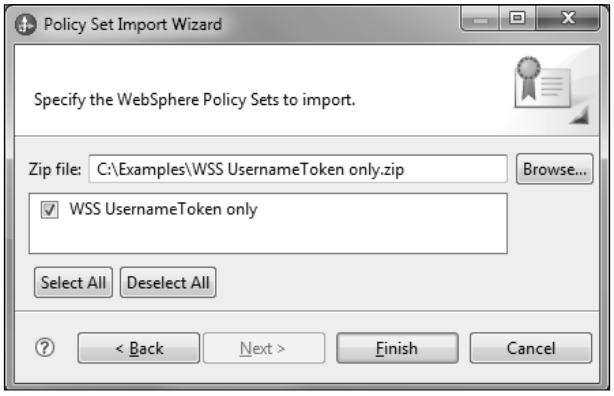



2. A dialog with a link to a ZIP file opens. Click on the link (or right-click and select **Save as …**) to download the file to a local disk:

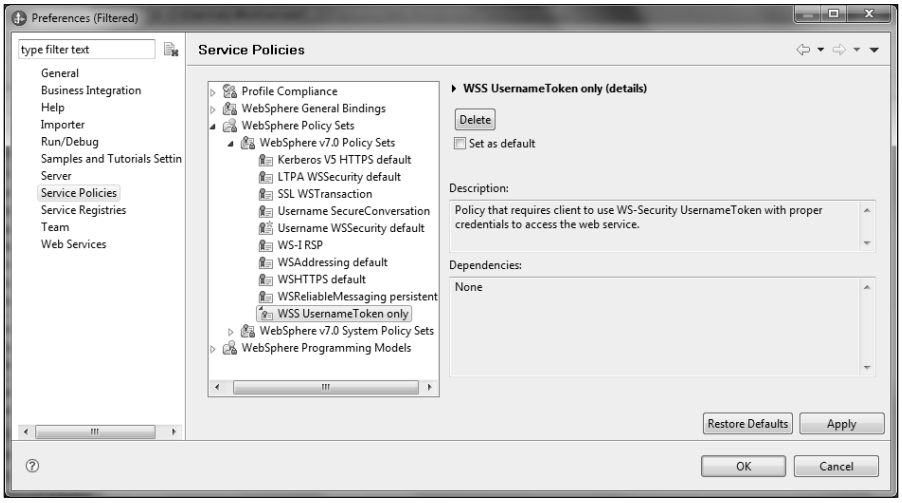


3. Next, we have to import the policy definitions to WebSphere Integration Developer. We will select **WebSphere Policy Sets** and click on **Next**:


4. Then we will select the ZIP file that we have created in the previous step. The policy set called **WSS UsernameToken only** is already selected for us to import. We will click on **Finish**:

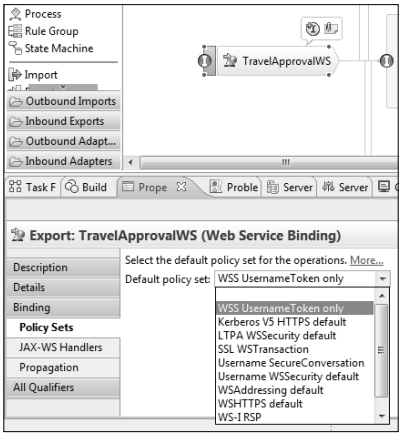


5. Now, we can check if the policy set has been imported properly. To do that, we have to go to the **Window | Preferences** menu. Select **Service Policies** in the navigation tree and expand **WebSphere Policy Sets | WebSphere v7.0 Policy Sets**. There we should see the **WSS UsernameToken only** policy listed. If we select it, we can see its details (description):




6. The policy set was successfully imported. To use it in the Assembly Diagram editor, we have to restart WebSphere Integration Developer.

7. All that is left here is to apply the imported policy set to our web service export. We will open the assembly diagram of our example. We will select the **TravelApprovalWS** web service export. In the **Properties** view, we will select the **Policy Sets** tab and in the **Default policy set:** drop-down menu, we select the **WSS UsernameToken only** policy:



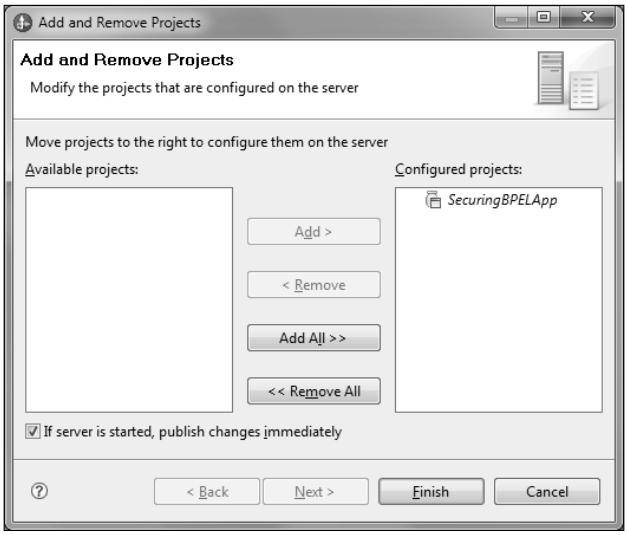
8. The last step is to save the changes made to the assembly diagram.

We have successfully defined that the **TravelApprovalWS** export uses the policy called **WSS UsernameToken only.** This way, we secured our process through its web service export. We managed to do that by exporting the **WSS UsernameToken only** policy set from the WebSphere Application Server console and importing it to the WebSphere Integration Developer. The policy set is defined on the server and referenced from our BPEL process module. This way, it is possible to reuse and modify the policy set during runtime.

# Testing a secured BPEL process

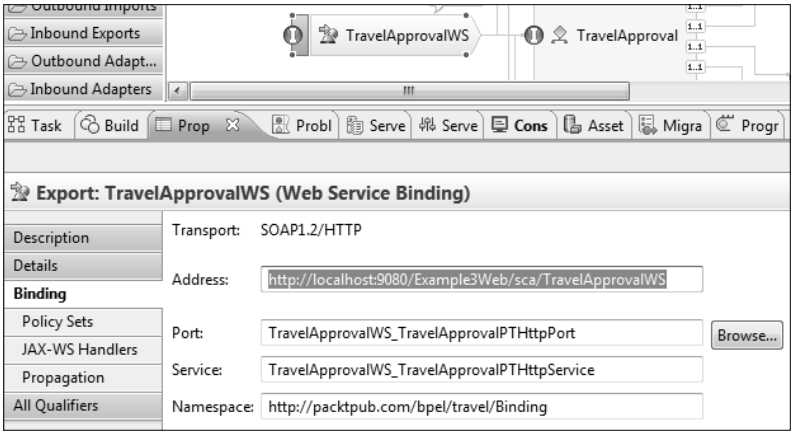
To test the secured BPEL process, we will deploy our solution on the server, locate the web service endpoint, create a test project, and then test the BPEL process through the web service export. First, we will try to invoke the BPEL process without providing credentials. Then we will invoke it with **UsernameToken** credentials.

1. First, we will deploy the solution to the server. Open **Add and Remove Projects** from the server context menu. Select **SecuringBPELApp** and click on **Add**. By default, **If server is started, publish changes immediately** is selected and, in this case, our example will be automatically published on the server. Otherwise, please select this checkbox. Click on **Finish**.



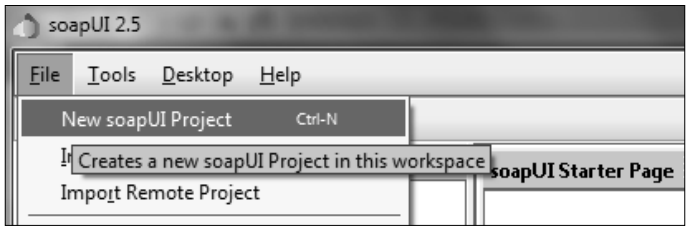

2. To test the service, we need to locate the **TravelApprovalWS** web service endpoint. We will use a free tool, soapUI, to test our service. We can download soapUI from `http://sourceforge.net/projects/soapui/files/`.

3. To locate the web service endpoint, we will select the export in the assembly diagram and in the **Properties** | **Binding** tab and copy the **Address:** field to the clipboard:

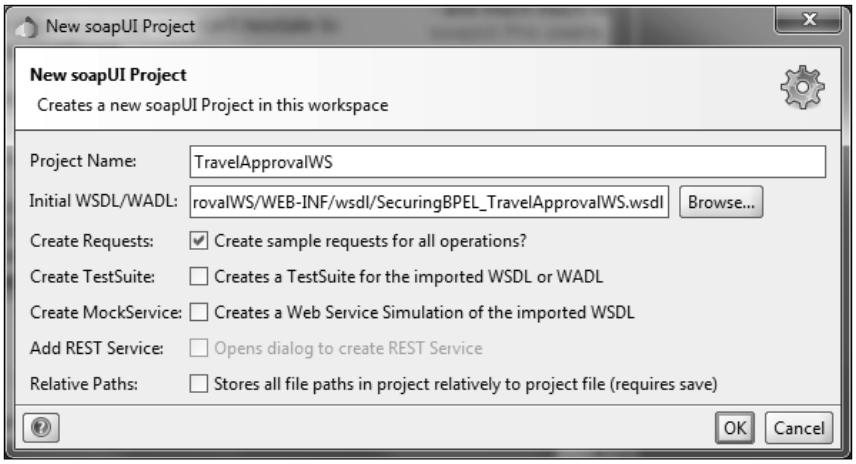



4. We can access the WSDL definition of our web service. This is the best way to create a test project in soapUI. We will paste the URL to the web browser address bar and add a `?wsdl` string at the end of the HTTP query string:


5. This way, the browser will redirect us to the absolute path of the WSDL file of our service. We need this address for the soapUI tool to call our BPEL process.
6. We are ready to create a test project in the soapUI. We will open the soapUI tool and create a new soapUI project with the WSDL URL we just located. In soapUI, we will select **File** | **New soapUI Project**:



7. To create the test project, we have to provide its name and WSDL location. For the **Project Name,** we will enter `TravelApprovalWS`, and in the **Initial WSDL/WADL** field, we will paste the WSDL URL from the clipboard. Leave other settings as they are and click on **OK**:

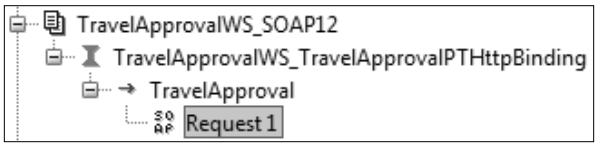



Thus, we have created a test project with a sample request to send to our BPEL process.

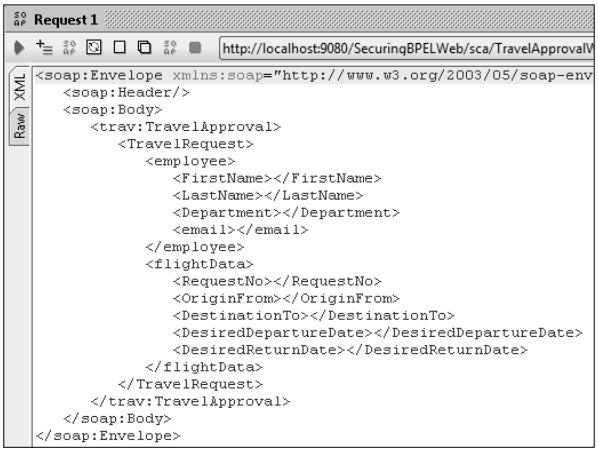
## Calling a BPEL process without credentials

First, we will try to call our BPEL process without specifying any credentials. This way we will see if the defined security mechanism through the WS-Policy definition works. We will test the security mechanism with the help of the following steps:

1. In the Project tree, we will expand the **TravelApprovalWS** project until we find the **Request 1** SOAP request leaf:



2. Double-clicking on **Request 1** will bring up the skeleton of the SOAP request that soapUI generated from the WSDL file. We can safely delete lines with comments stating that some elements are optional (all lines with `<!--Optional:-->`). Now try calling the BPEL process with an empty request by clicking the green triangle ( ) in the top-left corner of the screen.




3. We should get the following error:

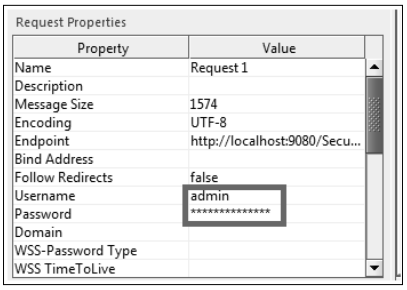
   **`Error message: A security token whose type is [http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken] is required.`**

We can see that we have secured the BPEL process. We cannot invoke it without security credentials. In the next section, we will provide security credentials in the request.

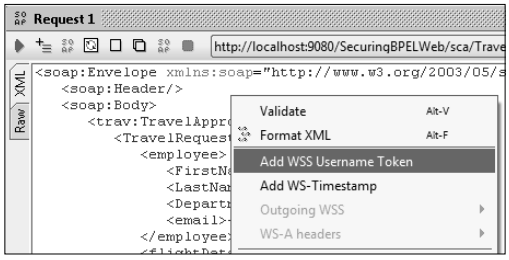
# Calling a BPEL process with credentials

Next, let us specify the credentials in the request and rerun the test by performing the following steps:

1. We will create a **UsernameToken** containing a username and password for authentication. We will enter the username and the password in the **Request Properties** view. We will use the default administrator username `admin` and password `admin`:

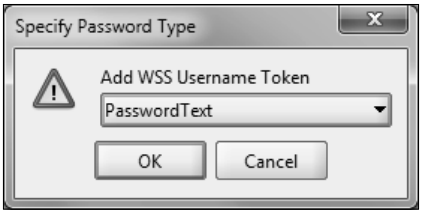



2. Now, we will add the WS-Security UsernameToken to the request by right-clicking on the request content and selecting **Add WSS Username Token**:



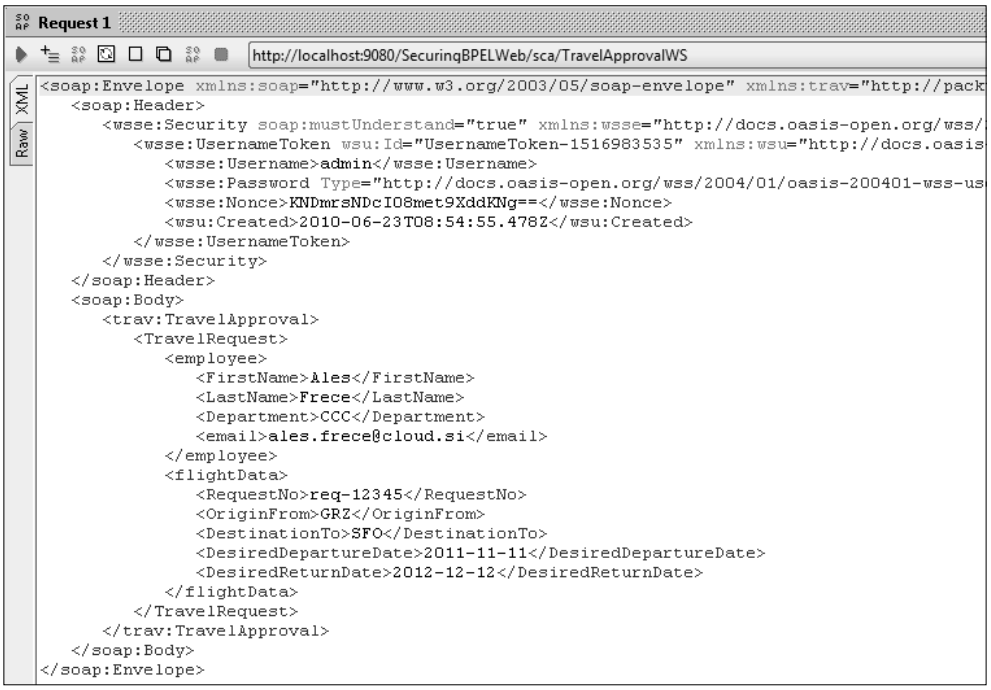

3. UsernameToken supports two types of passwords:
   - **PasswordText**: Plain text password
   - **PasswordDigest**: Base-64 SHA-1 hash value of the UTF-8 or equivalent encoded password

   The WS-Security specification states that the password digest offers no additional security over the plain text password without using secured transport protocol (such as HTTPS). This is one of the reasons why WebSphere Application Server does not support PasswordDigest. The other reason is that most advanced identity repositories (such as LDAP) use their own encryption/digest algorithms for passwords and many times it is impossible to get a password from the identity repository to create SHA-1 hash and do the comparison between hash from the identity repository and hash in the UsernameToken for authentication. So, we will choose the plain text **PasswordText** and click on **OK,** as shown in the following image.

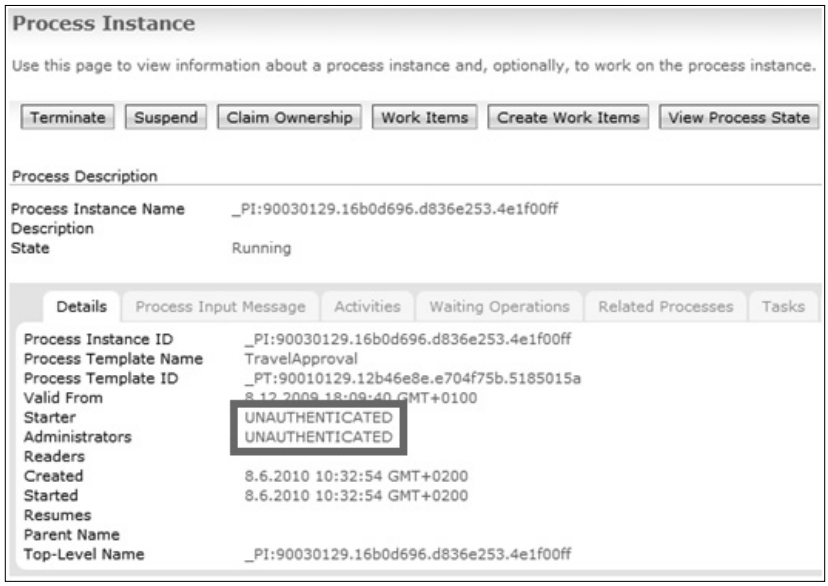


The `wsse:Security` header containing UsernameToken with username, password, nonce, and timestamp is inserted. We already mentioned timestamp earlier in this chapter. `wsse:Nonce`, on the other hand, might be new for us. It is a unique identifier in the request message to prevent replay attacks. See the next image for a nonce example.

4. Now, let us insert some test data into our request. Let us do that, because in the next step we will call our BPEL process with proper credentials and the BPEL process will not work correctly without the data specified.




5. If we try to run our BPEL process, we will see that it starts. We can see a new process instance in the Business Process Choreographer (`https://HOSTNAME(localhost):PORT(9443)/bpc`, or right-click on the server in the **Server** view and select **Launch | Business Process Choreographer Explorer**). We can locate one process instance of the **TravelApproval** process in the **Process Instances | Administrated by me** view.
6. If we click on that process instance, we can see its details. Be aware of **Starter** and **Administrators** properties where it states **UNAUTHENTICATED**. This is because so far we only authenticated the client when receiving the request message on the export. We have not propagated the client user identity to the BPEL process.




To propagate the user identity to the BPEL process, we have to define the Caller on the web service export. Caller definition ensures that successfully authenticated user identity is propagated to the succeeding service components within an assembly.

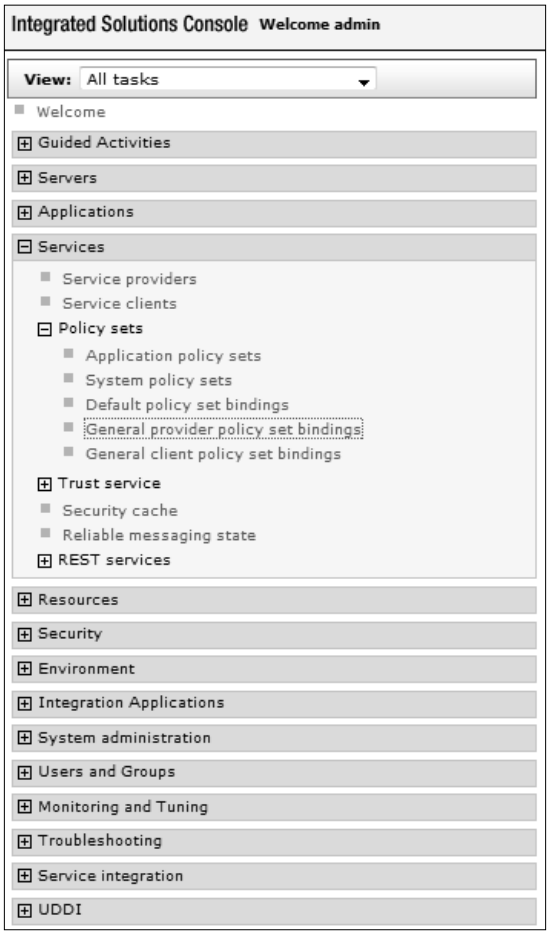
# Propagating user identity to a BPEL process

To propagate user identity from the **UsernameToken** to the BPEL process, we will first have to extract that identity from the **UsernameToken**. Then, we will propagate it to the BPEL process. We will enable identity extraction and propagation through policy set bindings. Policy set bindings will contain token consumer definition for identity extraction and caller definition for identity propagation.

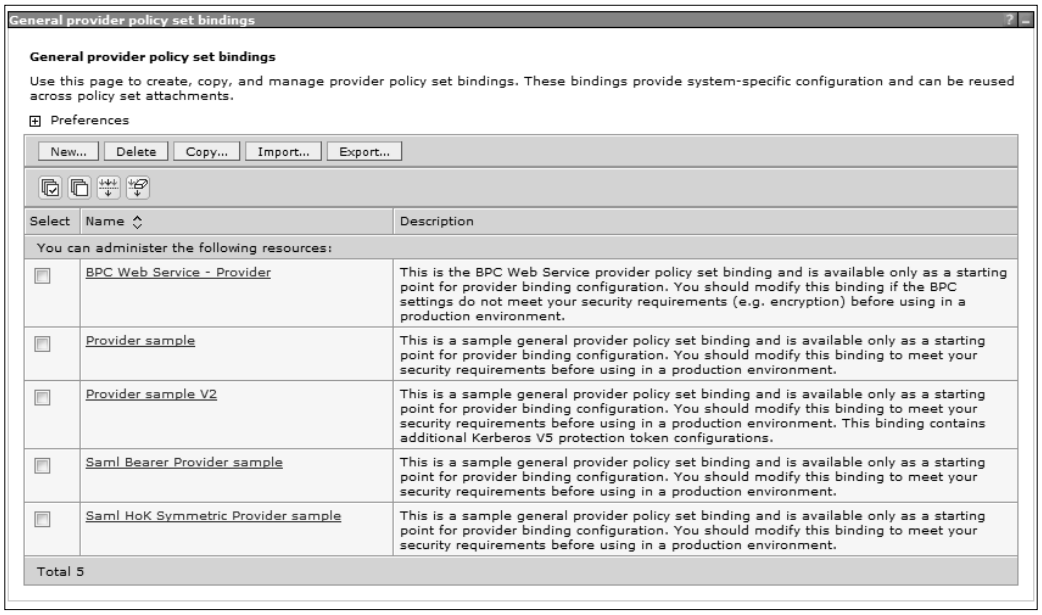
# Extracting user identity from UsernameToken

To enable the extraction of user identity from **UsernameToken**, we have to define a new policy set binding. We have to set this binding to use a **UsernameToken** consumer that will be performing the extraction. To achieve this, we have to complete the following steps:

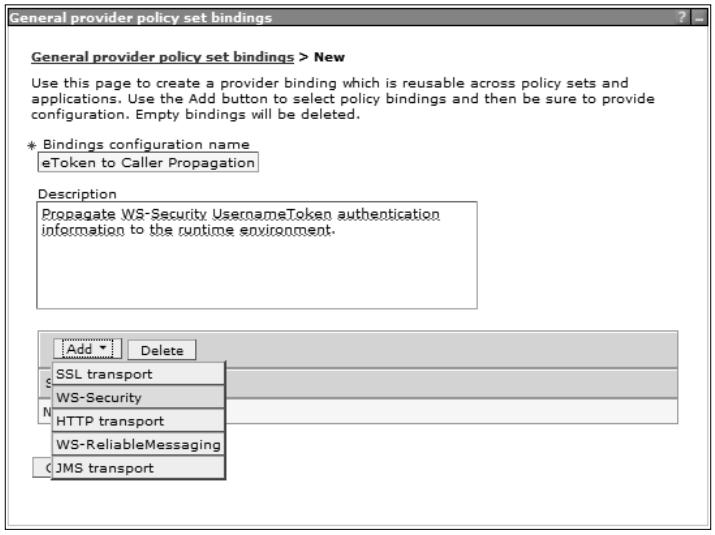
1. First, we have to create a new policy set binding. We will go to the **Integrated Solutions Console** and expand **Services** | **Policy Sets** | **General provider policy set bindings**:



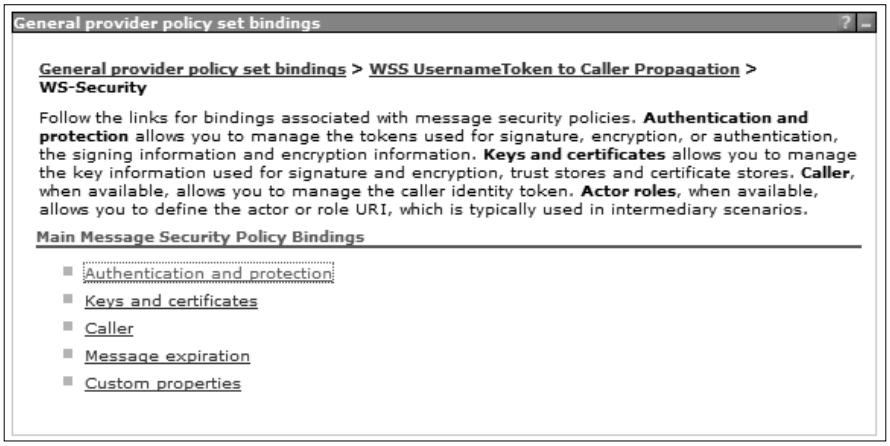

2. In the **General provider policy set bindings** list, we can see provided bindings. Apart from **BPC Web Service – Provider,** which is used for the Business Process Choreographer web service API, all bindings are only samples. Click on the **New…** button to create a new binding:




3. For the new binding, we need to provide a name, description (optional), and policies that the binding refers to (in our example, WS-Security). For the **Bindings configuration name**, we will enter `WSS UsernameToken to Caller Propagation` and `Propagate WS-Security UsernameToken authentication information to the runtime environment` for **Description**. Now, expand the **Add** drop-down menu and select **WS-Security** to add a WS-Security policy:



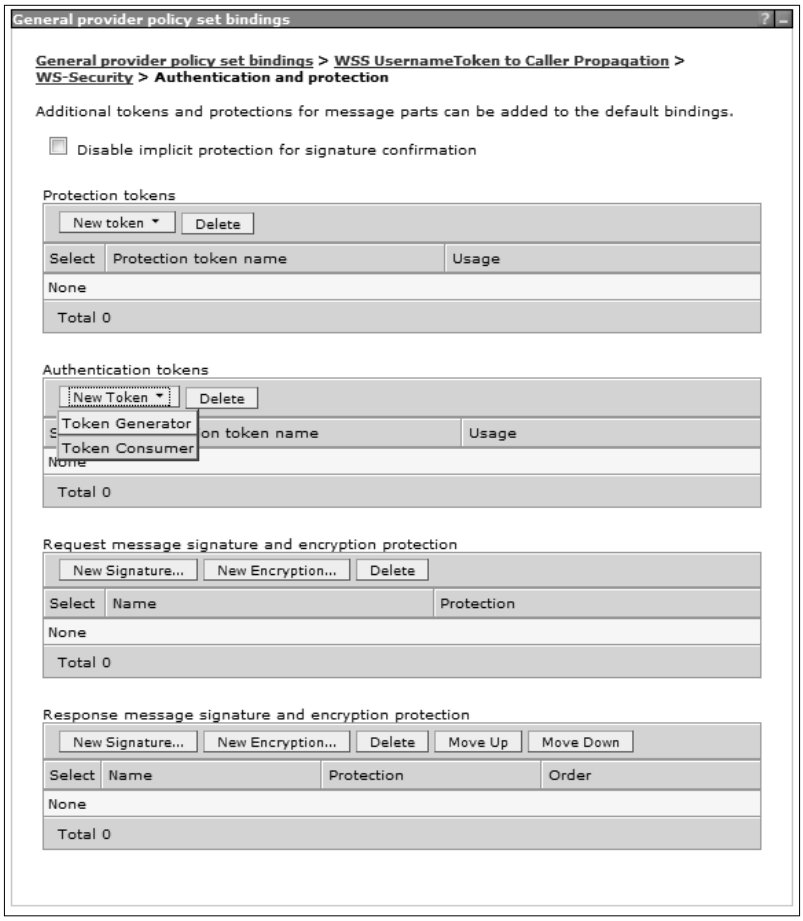
4. Next, we will define the token consumer for identity extraction. After clicking on the **WS-Security**, we will select the **Authentication and protection** link to define the UsernameToken consumer:




5. Under **Authentication tokens** we have:
   - **Protection tokens**, which sign messages to provide integrity or encrypt messages to provide confidentiality
   - **Authentication tokens**, which are used to provide or assert (propagate) the identity

   Detailed signing and encryption settings can be defined in the **Request message signature and encryption protection** and **Response message signature and encryption protection** sections. Going into details of these two groups of options exceeds the scope of this example.

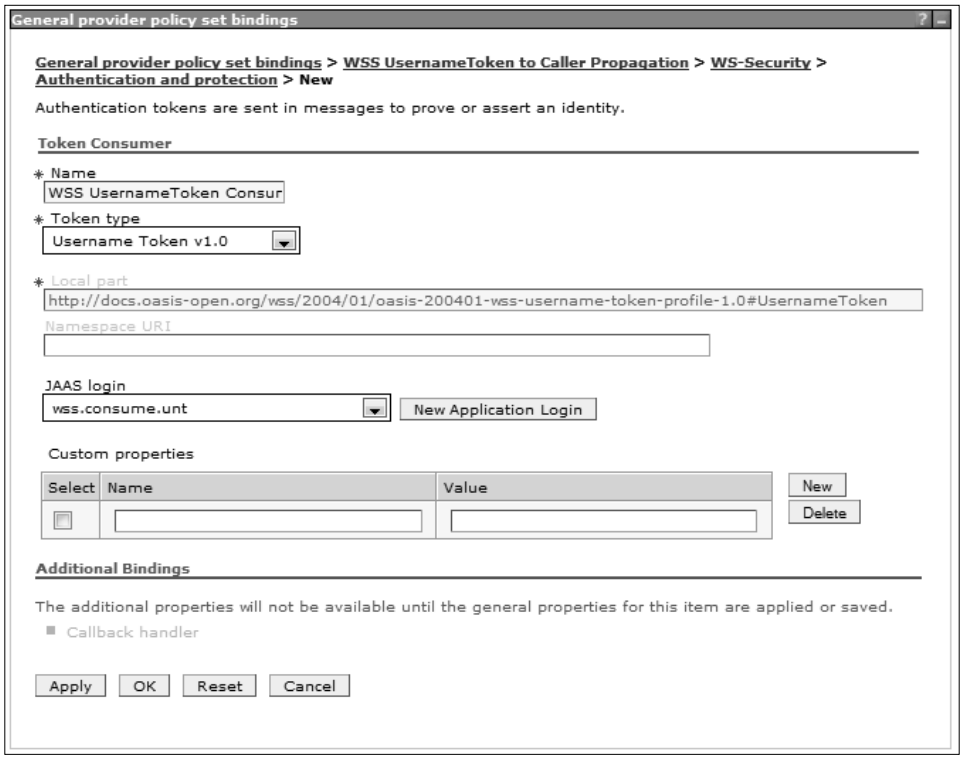
   To add a **UsernameToken** consumer, we will expand the **New Token** drop-down menu in **Authentication tokens** and select **Token Consumer**:



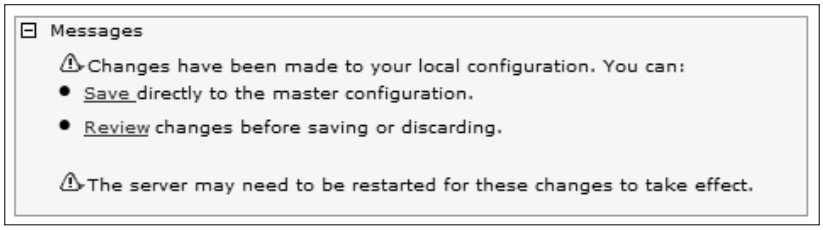

6. We created a new token consumer. If we want it to be the **UsernameToken** consumer, we have to specify its name, select **UsernameToken** for its type, and select the appropriate application login. Application login is basically a Java class that performs the actual authentication of the user based on his/her credentials provided in the token. So, for the token name, we will enter `WSS UsernameToken Consumer` and for its type, we will select **UsernameToken v1.0**. We will use version 1.0, as it is sufficient for our example. UsernameToken profile v1.1 contains some additional extensions that are used for deriving keys from the password for protecting message contents in the sense of integrity or confidentiality.

   After we have selected the **Token type**, the **Local part** field is automatically filled in for us. If we would have chosen some other authentication token (for example LTPA), **Namespace URI** would be automatically filled in for us.

We leave **JAAS login** at the default value (**wss.consume.unt**), because it is a default Java Authentication and Authorization Service system login for **UsernameToken** consumers. All described details are shown in the following screenshot:




7. Now, we should save changes made in the console to the master configuration, as shown in the following screenshot:



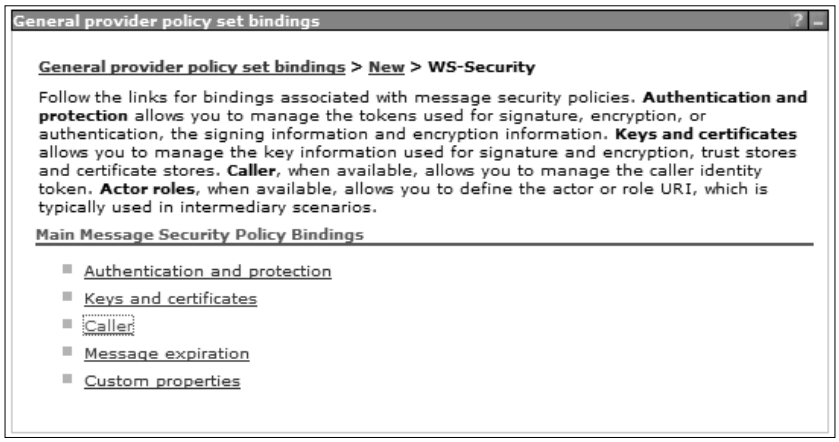

We have created a policy set binding with a **UsernameToken** consumer that can extract user identity from the token.

# Propagating an extracted user identity to a BPEL process

With the **UsernameToken** consumer we have defined a user identity extraction from the **UsernameToken**.

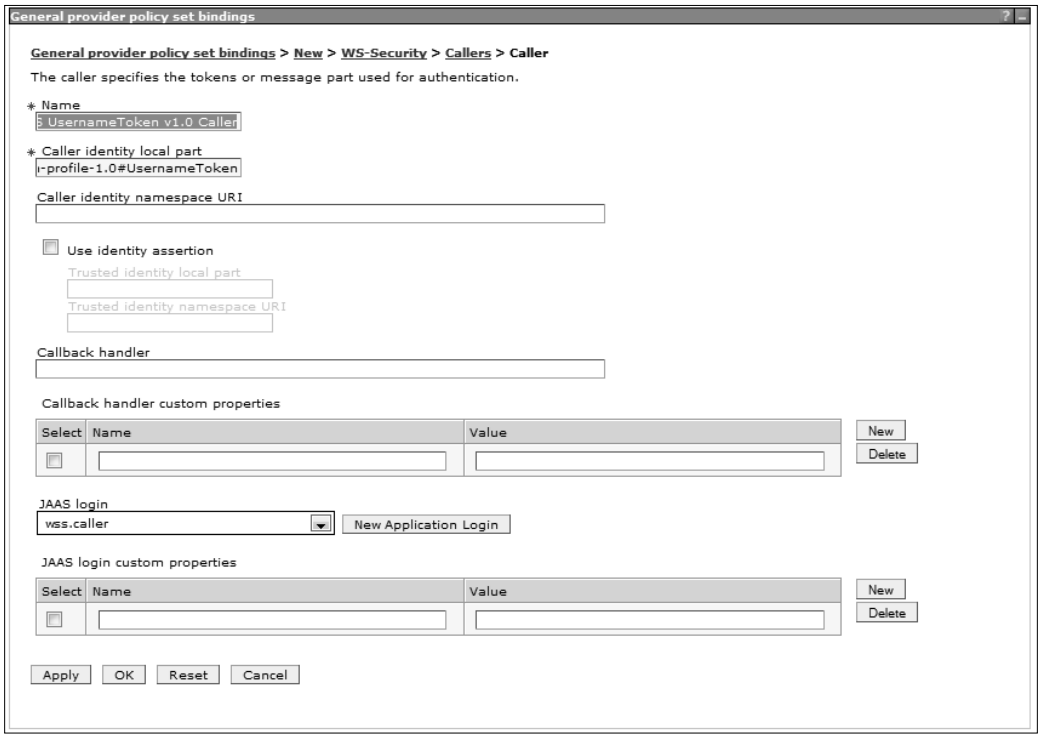
Now we have to define a **Caller** that will propagate the extracted identity to the succeeding components:

1. We will return to the **WS-Security** settings using the breadcrumb navigation and select the **Caller** link:




2. We will click on the **New** button to create a new Caller. We have to specify its name, identity local part (the URL from the token specification that tells from which type of token the identity should be propagated), and application login (the set of Java classes that takes care of token propagation).

3. For the **Name** enter `WSS UsernameToken v1.0 Caller` and for the **Caller identity local part** enter the URL `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken`. This is the URL defined in the **UsernameToken** profile v1.0 to uniquely identify **UsernameToken**. We will leave **JAAS login** at default (**wss.caller**), because it is a default Java Authentication and Authorization Service system login for the **UsernameToken** caller. Click on **OK** to finish creating the Caller.




We have successfully defined a policy set binding for propagating the identity from the **UsernameToken** to the succeeding components. We should now save the changes made through editors to the master configuration.

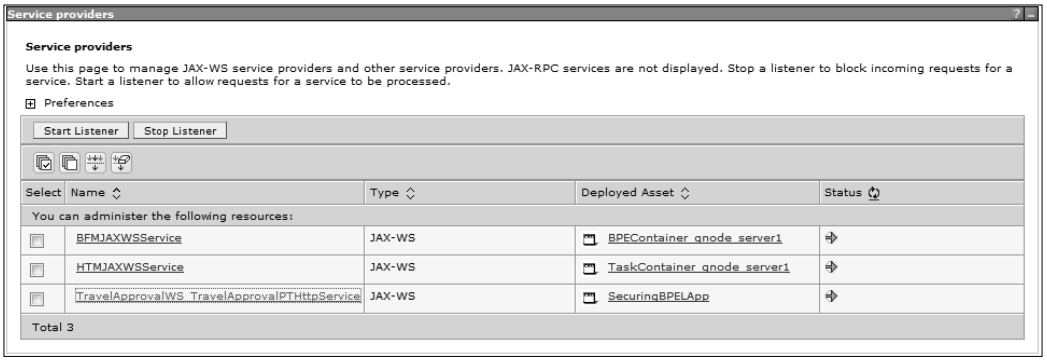
## Assigning a policy set binding to propagate an identity to a provider

All that is left to achieve the propagation of the user identity to the BPEL process is to assign the defined policy set binding to the web service that exports our BPEL process. This procedure is necessary after each (re)deploy of the BPEL process to the server:

1. Let us assign the policy set binding to the web service provider that represents the web service export of our BPEL process. In the **Integrated Solutions Console**, expand **Services**, and click on **Service providers**:


2. In the **Service providers** list we can see several providers. We will definitely see two: **BFMJAXWSService** and **HTMJAXWSService**. These two providers represent the already mentioned Business Process Choreographer web service APIs, for working with BPEL processes and human tasks respectively.

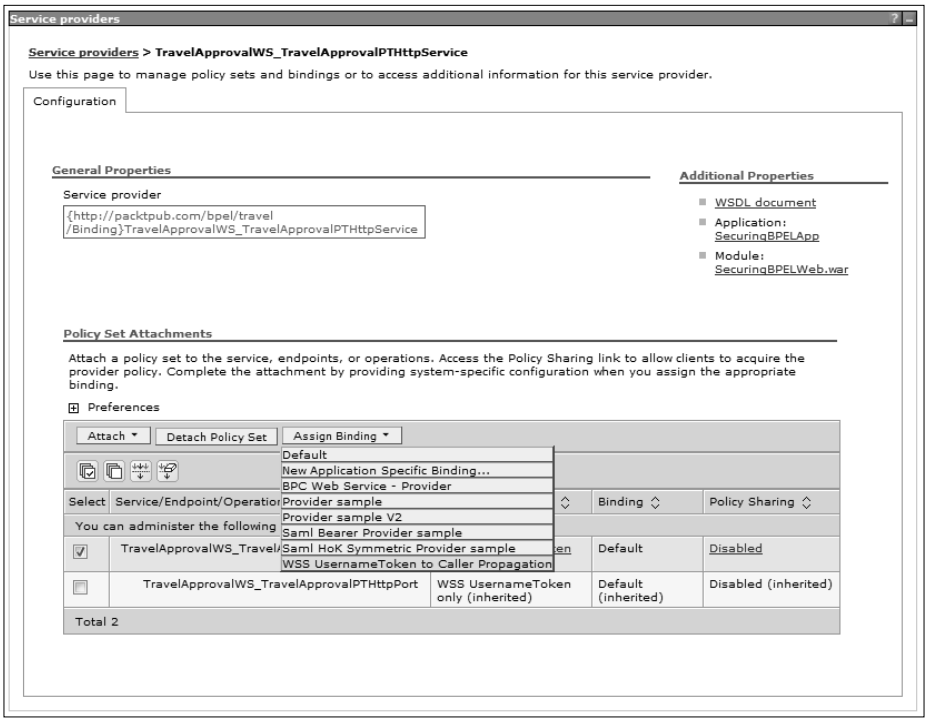
We will also see our **TravelApproval** provider. We will click on **TravelApprovalWS_TravelApprovalPTHttpService**:



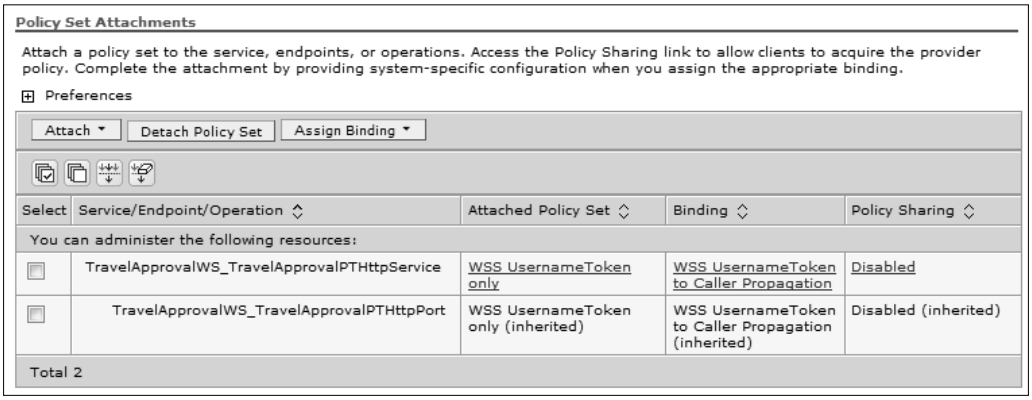

3. In **General Properties**, we can see a fully qualified XML name of the service. We can look at the service's WSDL document under **Additional Properties**. There we can also locate the application and the module of the service. In the **Policy Set Attachments** table, we can see the attached policies to this service provider. There is already the **WSS UsernameToken only** policy set attached to the provider. We achieved this through the definition in the assembly diagram earlier in this example.

   Now, we have to add the policy set binding that contains Caller to propagate a user identity to the BPEL process.

   We will click on the checkbox next to **TravelApprovalWS_ TravelApprovalPTHttpService**. We will expand the **Assign Binding** drop-down menu and select **WSS UsernameToken to Caller Propagation**:




4. With this, we have attached the policy set binding with the proper Caller defined and enabled the user identity propagation. Our configuration should look like the one shown in the following screenshot:



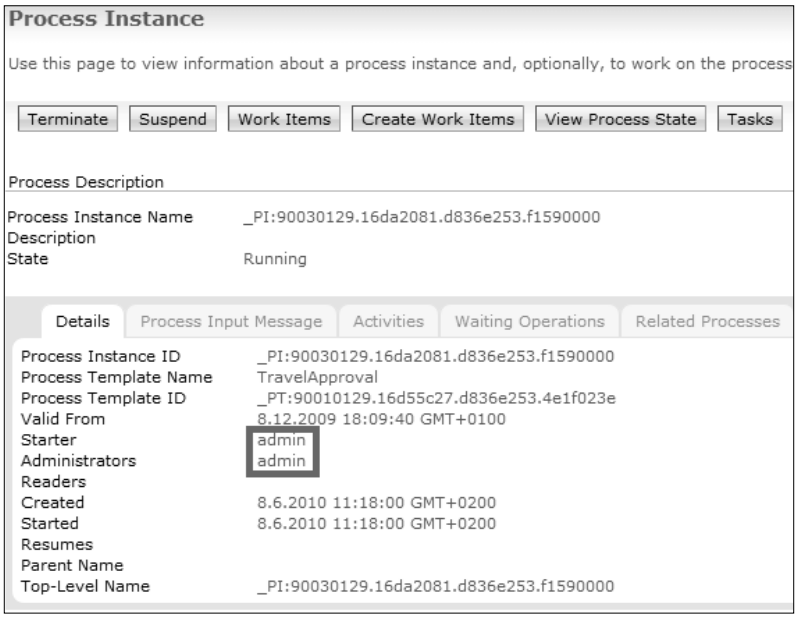

5. We will now save changes made to the master configuration.

So far, we have configured everything necessary for the propagation of the user identity to the BPEL process. For the configuration to start working in the previous steps, we assigned the defined policy set binding to the web service export of our BPEL process.

# Testing user identity propagation to BPEL process

Let us rerun the test case in soapUI to create another instance of the BPEL process. Now the process contains the user identity from the **UsernameToken**. Our BPEL process is now running in the name and ownership of the user that was authenticated at the web service export.

We can check if this is really the case. In the Business Process Choreographer, we will check the details of the created process instance. We will notice that the **Starter** and **Administrators** properties contain the identity of the user that called the web service:




# Restricting access to a BPEL process

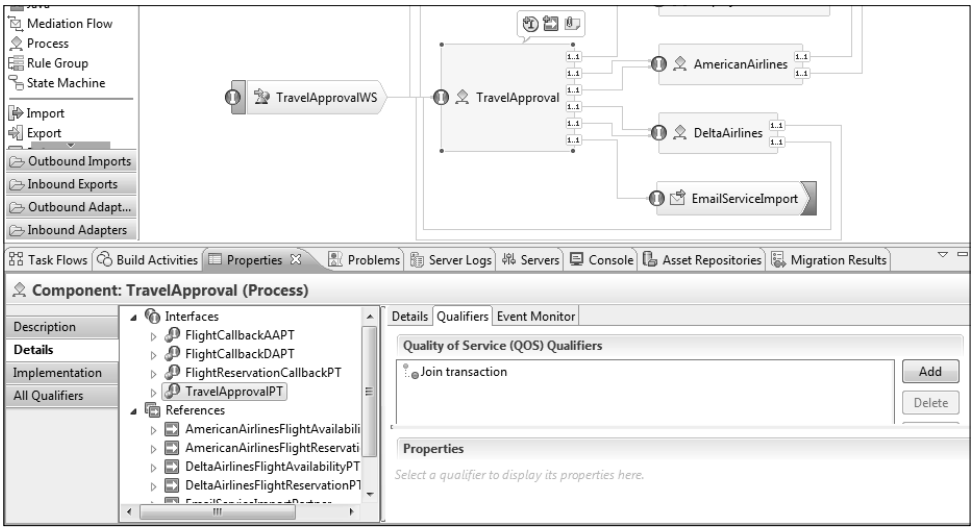
Now our BPEL process knows who has called it. All authenticated users can start a new instance of this process. The next thing that we will do is to restrict the group of users that can call our BPEL process. Only a specific group of users should be able to call our process. We can achieve this through the use of qualifiers.

# Setting a security permission qualifier

Qualifiers are means to alter behavior of an SCA component. Our BPEL process is an SCA component, so it is possible to define qualifiers to alter its behavior. For a discussion of qualifiers, please refer to *Chapter 4, BPEL Processes with IBM WebSphere*.

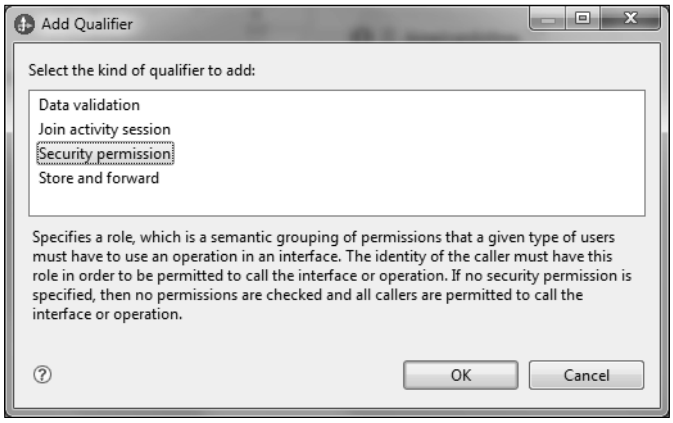
We will add a Security permission qualifier to our process. It will define the role name for the authenticated users. We will map this role to specific users in the deployment descriptor. At the runtime (on the server), it will be possible to add or remove users to/from this role. This way, we will be able to change a set of users that will have access to our process. To achieve this, we need to accomplish the following steps:

1. First, we have to define the Security permission qualifier on our BPEL process. To add the Security permission qualifier we have to select the BPEL process in the assembly diagram and bring up its properties. Select **Properties** | **Details** and then expand the **Interfaces** | **TravelApprovalPT** | **Qualifiers** tab on the right-hand side of the pane:




   There is already a **Join transaction** qualifier set. A join transaction qualifier tells whether the component will join the propagated transaction (value of `true`) from its client or not (value of `false`).

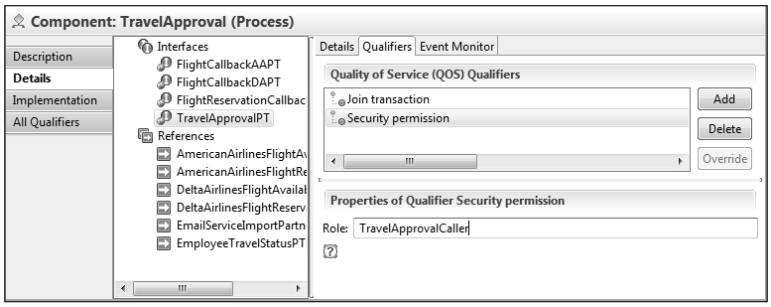
2. We will add another qualifier by clicking the **Add** button. A dialog opens and we select the **Security permission** qualifier. The **Security permission** qualifier on a component allows us to specify a role of users that have access to this component.




Other qualifiers that we can set are **Data validation** (turns on or off data validation in business object towards XSD schema), **Join activity session** (tells whether a component will join the propagated session from its client or not) and **Store and forward** (stores messages sent from the component in case of failure, for the administrator to be able to resubmit failed messages after the problem on the target component was fixed). Qualifiers are described in *Chapter 4*.

3. Next, we have to specify a role name. A role is a logical group of physical groups of users and users without a group assignment. A role defined through the Security permission qualifier restricts the users that can call the SCA component to this role only (direct assignment or indirect assignment through group membership).

Select the newly created **Security permission** qualifier. The **Properties of Qualifier Security permission** view pops up. Enter `TravelApprovalCaller` as **Role:**. Save the changes made to the assembly diagram:



We have defined the authorized role for calling our BPEL process. But this role does not contain any users yet. At this time, no user will be able to call our process. We will test if this holds true.

## Testing the authorization mechanism

Let us test if really no one can access our BPEL process at this time. We will try calling it with our administrative user (`admin`). Republish the changes made to the integration module to the server. After republishing is finished, we have to reconfigure a WS-Policy set binding for Caller, because it is lost during application redeployment. Please repeat the steps described in the section *Assigning a policy set binding to propogate an indentity to a provider* to rebind the WS-Policy set binding.

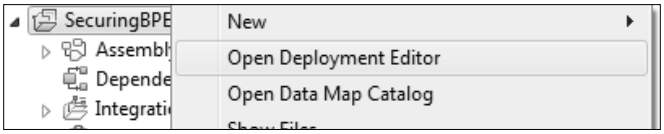
Try calling the web service export of our BPEL process from soapUI. The call does not work anymore, because the `admin` user is not in the `TravelApprovalCaller` role. We get the following exception:

```
Console Exception: Permission denied: User admin is not in
role:TravelApprovalCaller and does not have permission to invoke the
method
```

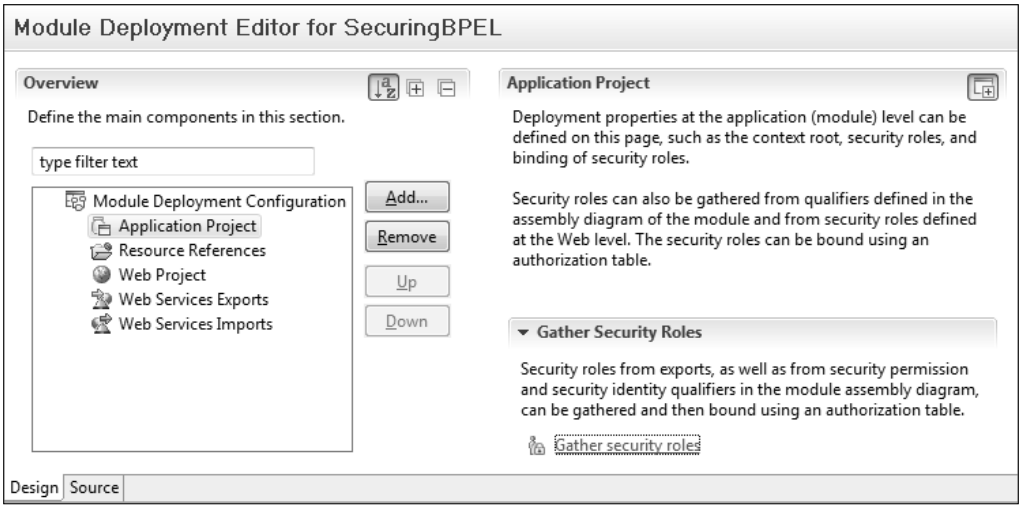
Now we will add the `admin` user to the `TravelApprovalCaller` role.

## Adding users to an authorized role

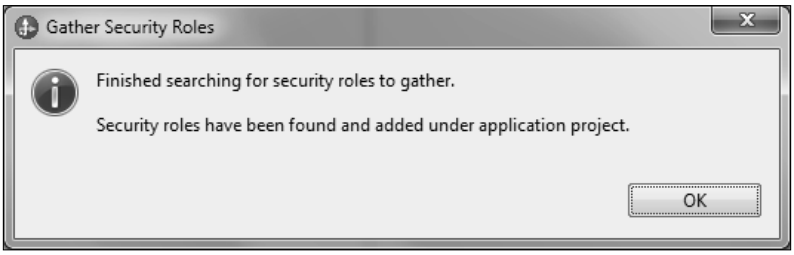
We have to map the `admin` username to the `TravelApprovalCaller` role. This mapping can be done in the deployment descriptor of our module/application. We will right-click on the **SecuringBPEL** module and select **Open Deployment Editor**:



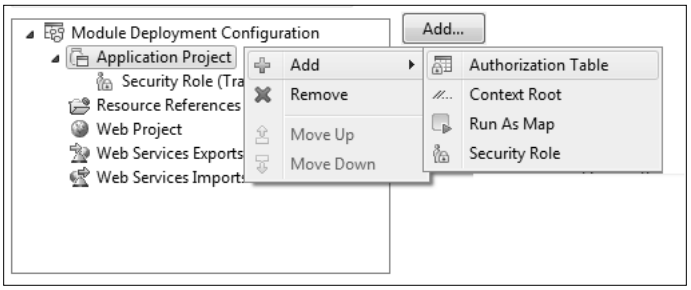
1. In **Module Deployment Editor for SecuringBPEL**, expand **Module Deployment Configuration | Application Project**. Expand **Gather Security Roles** and click on the **Gather security roles** link:



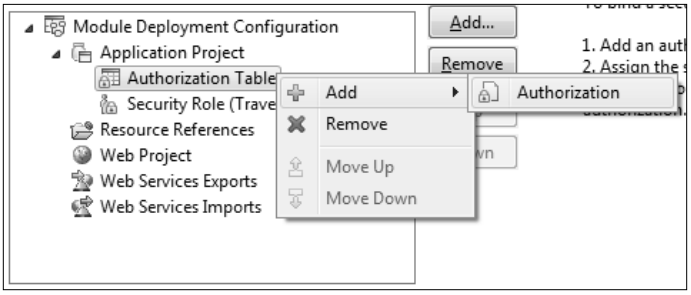

2. WebSphere Integration Developer gathers all role definitions from all over the module (for example, from all Security permission qualifiers). After it finishes, it displays a message as shown in the following screenshot:



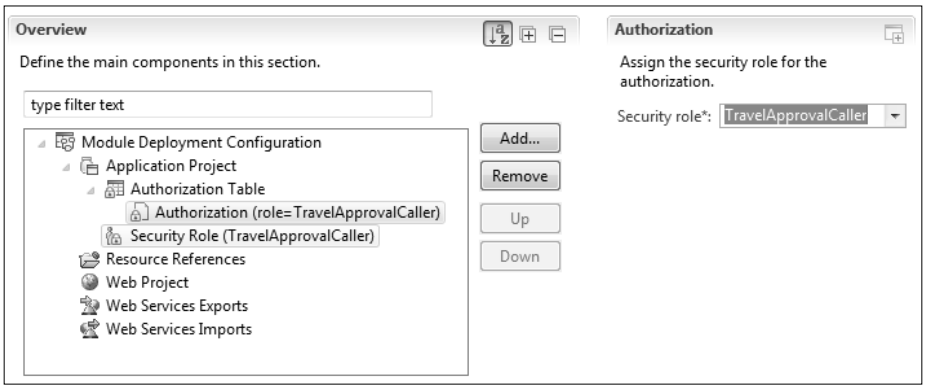

3. Click on **OK** and then expand **Application Project** to check that **Security Role (TravelApprovalCaller)** was added. Now, we will map the `admin` user to this role with the help of an authorization table. An authorization table contains all groups, users, and special subjects such as All authenticated users (everyone that is authenticated on the server) and Everyone (all authenticated users and the ones that are not).
4. Right-click on **Application Project** and select **Add | Authorization Table**:




5. An authorization table can contain multiple authorizations, each defining groups, users, and special subjects that map to one specific role. We will add one authorization that will provide the mapping for the `TravelApprovalCaller` role. Right-click on **Authorization Table**, and select **Add | Authorization**:

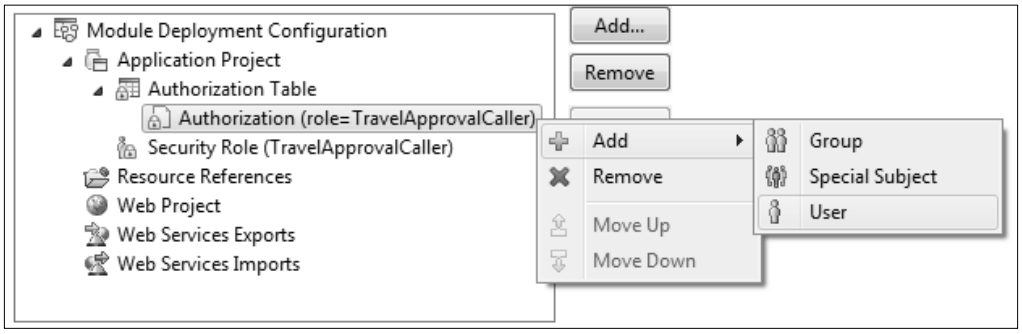



6. In our example, only one role has been gathered. Therefore, the **Security role:** in the **Authorization** section was automatically selected:




If we had more roles, it would be necessary to select the appropriate one from the **Security Role** drop-down menu.

7. Now we will map the `admin` user to this selected security role. Right-click on the **Authorization (role=TravelApprovalCaller)** and select **Add | User**.




8. In the **User** section, enter `admin` as **User name**:


We should save changes made in the Deployment Editor. After the solution redeploys, only `admin` should be able to call our BPEL process.

# Testing the authorization mechanism with an authenticated and authorized user

We have to redeploy our application to the server and invoke another request from soapUI to test if `admin` can start the BPEL process now. In the Business Process Choreographer, we can see that `admin` successfully started a new process instance.

# Summary

In this chapter, we have become familiar with the principles of securing BPEL processes. We have learned how to protect a BPEL process and how to achieve that it can be accessed by authenticated users only. We have demonstrated how to limit the access to the BPEL process to only those authenticated users that are also authorized to have access to the process.

We have shown how to expose a BPEL process as a web service and protect it to accept requests from authenticated users only. We achieved this through WS-Security UsernameToken, which should hold the username and password of a service consumer. We have shown how to propagate a caller's identity to the BPEL process instance. This way, the client user can be set as the process instance owner. Finally, yet importantly, we used identity propagation to restrict access to the BPEL process to the users that are authorized to access it. We achieved this with authorization rules that limit access to a BPEL process to specific users and groups only. We have shown how to apply authentication on the web service (WS-Security) and component level (SCA qualifier).

## Where to buy this book

You can buy WS-BPEL 2.0 for SOA Composite Applications with IBM WebSphere 7 from the Packt Publishing website: `https://www.packtpub.com/ws-bpel-2-0-for-soa-composite-applications-with-ibm-websphere-7/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.


**www.PacktPub.com**

**For More Information:**
**www.PacktPub.com/ws-bpel-2-0-for-soa-composite-applications-with-ibm-websphere-7/book**